

O'REILLY®

2-е издание



# Python для финансовых расчетов

---

ИСКУССТВО РАБОТЫ С ФИНАНСОВЫМИ ДАННЫМИ

Ив Хилпиш

2-Е ИЗДАНИЕ

# Python для финансовых расчетов



SECOND EDITION

# Python for Finance

---

Mastering Data-Driven Finance

*Yves Hilpisch*

Beijing • Boston • Farnham • Sebastopol • Tokyo **O'REILLY®**

2-Е ИЗДАНИЕ

# Python для финансовых расчетов

---

Искусство работы с финансовыми данными

*Ив Хилтун*

Київ  
Комп'ютерне видавництво  
"ДІАЛЕКТИКА"  
2021

УДК 336(075.8)

A95

Перевод с английского *И.В. Василенко*

Под редакцией *В.Р. Гинзбурга*

**Хилпиш, Ив.**

X45 Python для финансовых расчетов, 2-е изд./Ив Хилпиш; пер. с англ. И.В. Василенко. — Киев. : “Диалектика”, 2021. — 800 с. : ил. — Парал. тит. англ.

ISBN 978-617-7874-29-3 (укр.)

ISBN 978-1-492-02433-0 (англ.)

В новом издании книги разработчики и финансовые аналитики узнают, как применять различные инструменты Python для создания финансовых приложений и систем алгоритмической торговли. Все примеры книги написаны на Python 3 и доступны в виде интерактивных блокнотов Jupyter. Готовые программные решения помогут понять, как экосистема Python формирует технологический фундамент для финансовой индустрии.

УДК 336(075.8)

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly & Associates.

Copyright © 2021 by Dialektika Computer Publishing.

Authorized Russian translation of the English edition of *Python for Finance*, 2nd Edition (ISBN 978-1-492-02433-0) © 2019 Yves Hilpisch.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

ISBN 978-617-7874-29-3 (укр.)

ISBN 978-1-492-02433-0 (англ.)

© “Диалектика”, перевод, 2021

© 2019 Yves Hilpisch

Введение	19
Часть I. Python и финансовые вычисления	
Глава 1. Python как инструмент финансовых расчетов	27
Глава 2. Инфраструктура Python	63
Часть II. Основы Python	
Глава 3. Типы данных и структуры Python	93
Глава 4. Работа с массивами NumPy	121
Глава 5. Анализ данных с помощью библиотеки pandas	153
Глава 6. Объектно-ориентированное программирование	191
Часть III. Обработка и анализ финансовых данных	
Глава 7. Визуализация данных	215
Глава 8. Финансовые временные ряды	255
Глава 9. Операции ввода-вывода	283
Глава 10. Производительность Python	333
Глава 11. Математические инструменты	373
Глава 12. Стохастические методы	409
Глава 13. Статистический анализ	465
Часть IV. Алгоритмическая торговля	
Глава 14. Торговая платформа FXCM	545
Глава 15. Торговые стратегии	561
Глава 16. Автоматизированная торговля	607
Часть V. Анализ деривативов	
Глава 17. Принципы оценки опционов	649
Глава 18. Финансовое моделирование	665
Глава 19. Оценка деривативов	691
Глава 20. Оценка портфеля	717
Глава 21. Оценка на основе рыночных данных	739
Часть VI. Приложения	
Приложение А. Обработка значений даты и времени	765
Приложение Б. Класс опционов в модели Блэка — Шоулза — Мертона	781
Предметный указатель	787



---

# Содержание

Об авторе	17
Об изображении на обложке	17
<b>Введение</b>	<b>19</b>
Соглашения, принятые в книге	22
Файлы примеров к книге	23
Ждем ваших отзывов!	24
<hr/>	
<b>Часть I. Python и финансовые вычисления</b>	
<b>Глава 1. Python как инструмент финансовых расчетов</b>	<b>27</b>
Язык программирования Python	27
Краткая история Python	30
Экосистема Python	31
Круг пользователей Python	33
Стек научных пакетов	33
Технологии в финансовой отрасли	35
Инвестиции в технологии	35
Технологии как движущая сила	36
Технологии и кадры решают все	37
В погоне за скоростью, производительностью и объемами данных	38
Анализ в реальном времени	39
Python для финансовых расчетов	40
Синтаксис Python, применяемый в финансовых вычислениях	41
Эффективность и производительность кода Python	45
От прототипа к готовому приложению	51
Финансовые расчеты на основе данных и искусственного интеллекта	52
Финансовые системы, управляемые данными	52
Финансовые системы на основе искусственного интеллекта	57
Резюме	60
Дополнительные ресурсы	61
<b>Глава 2. Инфраструктура Python</b>	<b>63</b>
conda как менеджер пакетов	65
Установка Miniconda	65
Выполнение основных команд в менеджере conda	67

conda как менеджер виртуального окружения	72
Контейнеры Docker	76
Контейнеры и образы	76
Создание образа Docker с Ubuntu и Python	77
Облачные экземпляры	82
Открытый и закрытый ключи RSA	83
Конфигурационный файл Jupyter Notebook	84
Сценарий установки Python и Jupyter Notebook	86
Сценарий оркестровки для процесса установки дроплета	87
Резюме	89
Дополнительные ресурсы	90

---

## Часть II. Основы Python

<b>Глава 3. Типы данных и структуры Python</b>	<b>93</b>
Основные типы данных	94
Целые числа	94
Числа с плавающей точкой	95
Булевы значения	98
Строки	102
Пример: вывод и замена строк	104
Пример: регулярные выражения	107
Основные структуры данных	109
Кортежи	109
Списки	110
Пример: управляющие конструкции	112
Пример: функциональное программирование	114
Словари	116
Множества	117
Резюме	119
Дополнительные ресурсы	119
<b>Глава 4. Работа с массивами NumPy</b>	<b>121</b>
Массивы данных	122
Преобразование списка в массив	122
Класс array	124
Обычные массивы NumPy	127
Основные операции	127
Многомерные массивы	130
Метаинформация	134

Изменение формы и размера массива	135
Булевы массивы	139
Скорость выполнения операций	141
Структурированные массивы NumPy	143
Векторизация кода	145
Основные способы векторизации	145
Эффективное использование памяти	149
Резюме	151
Дополнительные ресурсы	152
<b>Глава 5. Анализ данных с помощью библиотеки pandas</b>	<b>153</b>
Класс <b>DataFrame</b>	154
Знакомство с классом <b>DataFrame</b>	154
Расширенные возможности класса <b>DataFrame</b>	159
Основные аналитические возможности	164
Основные инструменты визуализации	168
Класс <b>Series</b>	171
Группирование данных	172
Сложные операции извлечения данных	175
Конкатенация, соединение и слияние данных	179
Конкатенация	179
Соединение	181
Слияние	183
Производительность вычислений	186
Резюме	189
Дополнительные ресурсы	189
<b>Глава 6. Объектно-ориентированное программирование</b>	<b>191</b>
Обзор объектов Python	195
<b>int</b>	195
<b>list</b>	196
<b>ndarray</b>	197
<b>DataFrame</b>	199
Основные операции с классами Python	201
Модель данных Python	206
Код класса <b>Vector</b>	211
Резюме	212
Дополнительные ресурсы	212

---

## Часть III. Обработка и анализ финансовых данных

<b>Глава 7. Визуализация данных</b>	<b>215</b>
Статические двумерные графики	216
Одномерные наборы данных	217
Двухмерные наборы данных	223
Другие типы диаграмм	231
Статические трехмерные диаграммы	239
Интерактивные двумерные диаграммы	243
Базовые графики	243
Финансовые диаграммы	248
Резюме	252
Дополнительные ресурсы	253
<b>Глава 8. Финансовые временные ряды</b>	<b>255</b>
Финансовые данные	256
Импорт данных	256
Статистическая сводка	260
Изменения во времени	263
Прореживание данных	266
Скольльзящая статистика	268
Общие сведения	269
Пример технического анализа	271
Корреляционный анализ	274
Исходные данные	274
Логарифмическая доходность	276
Регрессионный анализ по методу наименьших квадратов	277
Корреляция	278
Высокочастотные данные	279
Резюме	282
Дополнительные ресурсы	282
<b>Глава 9. Операции ввода-вывода</b>	<b>283</b>
Базовые операции ввода-вывода в Python	284
Запись объектов на диск	285
Чтение и запись текстовых файлов	288
Работа с реляционными базами данных	292
Считывание и запись массивов NumPy	295
Ввод и вывод данных с помощью библиотеки pandas	299



Работа с реляционными базами данных	300
Импорт данных из реляционных баз данных	302
Работа с CSV-файлами	305
Работа с файлами Excel	306
Ввод и вывод данных с помощью PyTables	308
Работа с таблицами	308
Работа со сжатыми таблицами	317
Работа с массивами	319
Вычисления в условиях нехватки памяти	321
Ввод и вывод данных с помощью TsTables	324
Исходные данные	325
Хранение данных	326
Извлечение данных	328
Резюме	330
Дополнительные ресурсы	331
<b>Глава 10. Производительность Python</b>	<b>333</b>
Циклы	334
Python	335
NumPy	336
Numba	337
Cython	338
Алгоритмы	340
Простые числа	340
Числа Фибоначчи	345
Число $\pi$	349
Биномиальные деревья	353
Python	354
NumPy	355
Numba	357
Cython	358
Метод Монте-Карло	359
Python	361
NumPy	362
Numba	363
Cython	364
Параллельные вычисления	365
Рекурсивный алгоритм библиотеки pandas	366
Python	367
Numba	369

Cython	370
Резюме	371
Дополнительные ресурсы	372
<b>Глава 11. Математические инструменты</b>	<b>373</b>
Аппроксимация	374
Регрессия	375
Интерполяция	386
Выпуклое программирование	391
Глобальная оптимизация	392
Локальная оптимизация	394
Условная оптимизация	395
Интегрирование	398
Численное интегрирование	400
Интегрирование методами моделирования	400
Символьные вычисления	401
Общие сведения	401
Решение уравнений	404
Интегрирование	404
Дифференцирование	406
Резюме	407
Дополнительные ресурсы	408
<b>Глава 12. Стохастические методы</b>	<b>409</b>
Случайные числа	410
Моделирование	417
Случайные переменные	417
Случайные процессы	421
Уменьшение дисперсии	438
Оценка опционов	441
Европейские опционы	442
Американские опционы	448
Оценка рисков	451
Стоимость под риском	451
Поправка на кредитный риск	456
Общий сценарий Python	460
Резюме	463
Дополнительные ресурсы	464

<b>Глава 13. Статистический анализ</b>	<b>465</b>
Нормальное распределение	466
Эталонный портфель	467
Существующие исторические данные	479
Оптимизация портфеля	486
Данные	486
Теоретическое обоснование	488
Оптимальный портфель	493
Граница эффективности	496
Линия рынка капиталов	498
Байесовская статистика	502
Формула Байеса	502
Байесовская регрессия	503
Два финансовых инструмента	508
Обновление оценочных значений со временем	513
Машинное обучение	518
Обучение без учителя	519
Обучение с учителем	522
Резюме	540
Дополнительные ресурсы	541

---

## **Часть IV. Алгоритмическая торговля**

<b>Глава 14. Торговая платформа FXCM</b>	<b>545</b>
Настройка программного интерфейса FXCM	546
Получение данных	547
Получение тиковых данных	548
Получение свечных данных	550
Работа с программным интерфейсом FXCM	553
Получение исторических данных	553
Получение потоковых данных	556
Размещение заявок	557
Учетные данные	559
Резюме	560
Дополнительные ресурсы	560
<b>Глава 15. Торговые стратегии</b>	<b>561</b>
Простое скользящее среднее	562
Импорт данных	563

Торговая стратегия	564
Векторизованное тестирование на исторических данных	566
Оптимизация	569
Гипотеза случайного блуждания	571
Линейная регрессия по методу наименьших квадратов	575
Данные	575
Регрессия	578
Кластеризация	581
Частотный подход	583
Классификация	586
Два бинарных признака	586
Пять бинарных признаков	588
Пять дискретизированных признаков	590
Последовательное разделение данных на обучающий и тестовый наборы	592
Рандомизированное разделение данных на обучающий и тестовый наборы	594
Глубокие нейронные сети	596
DNN и библиотека Scikit-learn	596
DNN и библиотека TensorFlow	599
Резюме	604
Дополнительные ресурсы	604
<b>Глава 16. Автоматизированная торговля</b>	<b>607</b>
Управление капиталом	608
Критерий Келли в биномиальной модели	608
Критерий Келли в биржевой торговле	614
Торговая стратегия, основанная на машинном обучении	619
Векторизованное тестирование на исторических данных	620
Оптимальный леверидж	626
Анализ рисков	628
Сохранение объекта модели	633
Веб-алгоритм	633
Инфраструктура и развертывание	636
Протоколирование и мониторинг	638
Резюме	640
Сценарии Python	641
Автоматизированная торговая стратегия	641
Мониторинг стратегии	644
Дополнительные ресурсы	645



---

## Часть V. Анализ деривативов

<b>Глава 17. Принципы оценки опционов</b>	<b>649</b>
Фундаментальная теорема ценообразования финансовых активов	650
Простой пример	650
Общая модель	651
Риск-нейтральное дисконтирование	653
Моделирование и обработка дат	653
Постоянная краткосрочная ставка	656
Рыночная среда	658
Резюме	662
Дополнительные ресурсы	663
<b>Глава 18. Финансовое моделирование</b>	<b>665</b>
Генерирование случайных чисел	666
Общий класс моделирования	668
Геометрическое броуновское движение	673
Класс моделирования	673
Пример использования	676
Прыжковая диффузия	679
Класс моделирования	679
Пример использования	682
Диффузия по закону квадратного корня	684
Класс моделирования	685
Пример использования	687
Резюме	689
Дополнительные ресурсы	690
<b>Глава 19. Оценка деривативов</b>	<b>691</b>
Общий класс оценки деривативов	692
Европейский опцион	696
Класс оценки	697
Пример использования	699
Американский опцион	705
Метод наименьших квадратов Монте-Карло	705
Класс оценки	707
Пример использования	710
Резюме	713
Дополнительные ресурсы	715

<b>Глава 20. Оценка портфеля</b>	717
Деривативные позиции	718
Класс деривативной позиции	718
Пример использования	721
Портфели деривативов	722
Класс портфеля	723
Пример использования	728
Резюме	736
Дополнительные ресурсы	738
<b>Глава 21. Оценка на основе рыночных данных</b>	739
Данные опционов	740
Калибровка модели	743
Релевантные рыночные данные	743
Моделирование опционов	745
Процедура калибровки	748
Оценка портфеля	755
Моделирование опционных позиций	755
Портфель опционов	756
Код Python	758
Резюме	760
Дополнительные ресурсы	761
<hr/>	
<b>Часть VI. Приложения</b>	
<b>Приложение А. Обработка значений даты и времени</b>	765
Python	765
NumPy	771
pandas	775
<b>Приложение Б. Класс опционов в модели Блэка — Шоулза — Мертона</b>	781
Определение класса	781
Пример использования	783
<b>Предметный указатель</b>	787



## Об авторе

---

**Ив Хилпиш** — основатель проекта The Python Quants (<https://home.tpq.io/>), ориентированного на применение программных продуктов с открытым исходным кодом в финансовом анализе, искусственном интеллекте и алгоритмической торговле. Он также возглавляет компанию The AI Machine (<http://aimachine.io/>), которая занимается разработкой стандартизированных систем алгоритмической торговли на базе искусственного интеллекта.

Помимо этого Ив Хилпиш читает курс финансовой инженерии в рамках проекта CQF ([www.cqf.com](http://www.cqf.com)) и руководит первой обучающей онлайн-программой, нацеленной на получение “университетского сертификата по алгоритмической торговле на Python” (<http://certificate.tpq.io/>).

Ив Хилпиш написал библиотеку DX Analytics (<http://dx-analytics.com/>), представляющую собой пакет эффективных инструментов финансового анализа. Он проводит конференции и семинары, посвященные искусственному интеллекту, финансовому анализу и алгоритмической торговле, по всему миру.

## Об изображении на обложке

---

Животное, изображенное на обложке книги, — это *гаитянский щелезуб* (лат. *Solenodon paradoxus*), исчезающий вид млекопитающих, который в живой природе встречается исключительно на острове Гаити. Его ареал обитания охватывает в основном Доминиканскую Республику и в незначительной степени — Республику Гаити.

Питается гаитянский щелезуб преимущественно членистоногими, червями, улитками и пресмыкающимися, но в его рацион также входят корни, фрукты и листья. Вес взрослой особи составляет около килограмма, длина тела — 30–35 см, длина хвоста — 20–25 см. Внешне напоминает крупную землеройку. Все его тело, за исключением лап, хвоста и мордочки, покрыто рыжей шерстью, которая заметно светлее на животе, чем на спине.

Щелезуб ведет скрытный, преимущественно ночной образ жизни. Днем спит в естественных укрытиях или в норах. Двигается неуклюже, но бежит довольно быстро. Как и любые другие ночные животные обладает прекрасным слухом, обонянием и осязанием. У щелезуба очень характерный мускусный запах.

Гаитянские щелезубы ядовиты: токсичная слюна, которая парализует жертву, выделяется подчелюстной слюнной железой и впрыскивается через глубокую бороздку второго нижнего резца. Примечательно, что они не имеют иммунитета к собственному яду и могут погибнуть от укусов, полученных во время драк между собой. Образовав устойчивые социальные отношения, щелезубы подолгу проживают небольшими семьями на одном месте.

Многие из животных, изображаемых на обложках книг издательства O'Reilly, находятся под угрозой вымирания, и все они представляют ценность для нашего мира. Чтобы узнать о том, каким может быть ваш личный вклад в их спасение, посетите сайт [animals.oreilly.com](http://animals.oreilly.com).

Изображение на обложке взято из книги *Illustrated Natural History* Джона Георга Вуда.

В наши дни Python является одним из ключевых инструментов разработки стратегических технологий в финансовой сфере. Когда я приступил к написанию первого издания книги в 2013 году, мне приходилось проводить множество бесед и презентаций, бесконечно убеждая скептиков в конкурентных преимуществах Python как языка разработки финансовых приложений по сравнению с другими языками программирования и платформами. Спустя пять лет уже ни у кого не осталось никаких сомнений: финансовые учреждения по всему миру активно применяют Python и его разветвленную экосистему пакетов анализа данных, визуализации и машинного обучения.

Помимо финансовой сферы Python зачастую является языком выбора в курсах изучения программирования. Причина заключается не только в понятном синтаксисе и поддержке множества парадигм, но и в наличии продвинутых средств разработки приложений в таких областях, как искусственный интеллект, машинное обучение и глубокое обучение. Самые популярные пакеты и библиотеки для этих областей либо написаны непосредственно на Python (как, например, Scikit-learn), либо содержат оболочки, написанные на Python (например, TensorFlow).

Финансовая отрасль сама по себе вступает в новую эпоху, определяемую двумя движущими факторами. В первую очередь это появление программного доступа практически ко всем доступным финансовым данным. Причем такой доступ возможен в режиме реального времени, что позволяет строить финансовые системы, управляемые данными. В былые времена большинство торговых и инвестиционных решений принималось трейдерами или финансовыми менеджерами на основании информации, полученной через СМИ или личные контакты. С появлением компьютеров стало возможным просматривать текущую биржевую информацию на экране монитора, но в современном мире ежеминутно генерируется такой объем финансовых данных, что человеку за ним просто не поспеть. Справиться с обработкой нескончаемого потока финансовой информации способны только компьютеры. Как следствие, большинство операций по торговле финансовыми активами управляется программами, а не трейдерами.

Вторым фактором стала всевозрастающая роль искусственного интеллекта в финансовых расчетах. Все больше финансовых учреждений применяет алгоритмы машинного и глубокого обучения в операционной деятельности и в системах принятия инвестиционных решений. Первая специализированная книга, посвященная применению машинного обучения в финансах, вышла в начале 2018 года, и за ней последовала лавина других подобных книг. Все это ведет к появлению *финансовых систем на основе искусственного интеллекта*, в которых гибкие, параметризуемые алгоритмы машинного и глубокого обучения замещают традиционную финансовую теорию, эффективную в прошлом, но не способную справиться с вызовами новой эпохи.

Python со своей экосистемой как нельзя лучше подходит на роль ведущего языка программирования для разработки современных финансовых решений. Несмотря на то что в книге будут рассматриваться базовые алгоритмы машинного обучения (и нейронных сетей), основной акцент сделан на инструментах обработки и анализа данных. Чтобы в полном объеме охватить тему применения искусственного интеллекта в финансах, пришлось бы написать отдельную книгу. В то же время методы искусственного интеллекта требуют наличия столь огромных объемов данных, что в первую очередь следует научиться управлять данными.

Второе издание книги было существенно переработано. В частности, добавилась часть IV, посвященная алгоритмической торговле, которая набирает популярность в финансовой индустрии. Была также расширена часть II, посвященная описанию основных инструментов Python, которые будут задействованы в последующих частях. Наряду с этим из нового издания было удалено несколько глав, посвященных веб-пакетам типа Flask, которые достаточно подробно описаны в специализированной литературе.

Во втором издании рассмотрено большое число тем, связанных с финансовыми вычислениями. Основное внимание уделяется инструментам Python, применяемым для работы с финансовыми данными. Как и в первом издании, подход носит прикладной характер, поскольку предпочтение отдается программной реализации и представлению результатов, а не теоретическим выкладкам. При этом мы будем стараться увидеть общую картину, а не концентрироваться на параметрах отдельных классов, методов или функций.

Следует подчеркнуть, что книга не является ни введением в программирование на Python, ни учебником по финансам в целом. Каждая из этих тем подробно рассматривается во множестве великолепных руководств. Книга находится как бы на стыке двух областей, поэтому от читателя предполагается наличие базовых знаний по программированию (не обязательно на Python) и

финансам. Книга поможет узнать, как применять Python и его библиотеки для решения финансовых задач.

Блокноты Jupyter и примеры программ, используемые в книге, доступны на авторском сайте Quant Platform (<http://py4fi.pqr.io>), на котором можно зарегистрироваться бесплатно.

Компания The Python Quants предлагает множество ресурсов, посвященных применению Python в финансовом анализе, искусственном интеллекте и алгоритмической торговле. Для начала посетите следующие сайты:

- сайт компании The Python Quants (<https://tpq.io/>);
- сайт автора книги (<http://hilpisch.com/>);
- сайт, посвященный книгам автора (<https://books.tpq.io/>);
- сайт, посвященный авторским онлайн-курсам (<https://training.tpq.io/>);
- сайт программы сертификации (<https://certificate.tpq.io/>).

Среди всех проектов, реализованных автором за последние несколько лет, самое большое достижение — это программа сертификации специалистов по алгоритмической торговле на Python. Участникам программы предлагается более 150 лекционных часов, 1200 страниц документации, 5000 строк кода Python и 60 блокнотов Jupyter. Набор в программу (к слову, постоянно обновляемую и дополняемую новыми курсами) проводится несколько раз в год. Это первая онлайн-программа такого рода, в которой выпускникам выдается официальный университетский диплом в сотрудничестве с Саарским университетом прикладных наук (<http://htwsaar.de/>).

Помимо этого автор запустил платформу The AI Machine (<https://aimachine.io/>), предназначенную для внедрения автоматизированных систем алгоритмической торговли. В рамках данного проекта мы стремимся реализовать все то, чему обучали студентов за последние годы. В наши дни благодаря Python и технологиям искусственного интеллекта подобные проекты становятся возможными даже для таких небольших команд, как наша.

Введение к первому изданию книги заканчивалось такими словами.

Мне необычайно приятно осознавать, что Python зарекомендовал себя передовой технологией в финансовой сфере. Убежден, в будущем он станет играть еще более важную роль в таких областях, как анализ рисков и высокопроизводительные вычисления. Надеюсь, книга поможет профессиональным разработчикам, ученым и студентам применять Python для решения самых сложных задач.



В далеком 2014 году сложно было представить, насколько широко Python будет применяться в финансовых расчетах. Спустя пять лет стоит признать, что действительность превзошла самые смелые ожидания. Хочется верить, что первое издание книги внесло свою лепту в продвижение Python. В любом случае стоит сказать огромное спасибо всем разработчикам программного обеспечения с открытым исходным кодом за их неутомимый труд, благодаря которому сказка стала былью.

## Соглашения, принятые в книге

В этой книге приняты следующие условные обозначения.

### *Курсив*

Служит для выделения ключевых терминов, которые следует знать.

### Моноширинный шрифт

Используется для оформления листингов, а также для выделения имен файлов, пакетов и программных элементов, в частности операторов, переменных, ключевых слов и т.п. Этим же шрифтом выделяются URL-адреса.

### Моноширинный курсив

Применяется для выделения программных элементов, которые должны вводиться пользователем или заменяться значениями по контексту.



Этой пиктограммой помечаются советы или рекомендации.



Этой пиктограммой помечаются примечания к основному тексту.



Этой пиктограммой помечаются предупреждения, на которые следует обратить внимание.

## Файлы примеров к книге

Все примеры программ, используемые в книге (в частности, блокноты Jupyter и исходные коды сценариев и программных модулей Python), доступны на сайте книги:

<http://py4fi.pqr.io>

Пройдя бесплатную регистрацию, вы получите доступ к программной среде, в которой можно как скачать все примеры для самостоятельного развертывания, так и запустить каждый блокнот Jupyter прямо на сайте. Все блокноты глав доступны в разделе **Notebooks**<sup>1</sup>. Доступ к файловому хранилищу можно получить с помощью команды **Tools⇒File Manager**. Инструкции по работе с сайтом содержатся в видеоролике, который вызывается по команде **Help⇒Platform**.

---

<sup>1</sup> В ряде случаев примеры, приводимые в книге, незначительно отличаются от кода, содержащегося в блокнотах. Изменения в основном связаны с необходимостью получения корректных версий рисунков. При возникновении спорных ситуаций сверяйтесь с кодом, приведенным в книге — *Примеч. ред.*

## Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: [info.dialektika@gmail.com](mailto:info.dialektika@gmail.com)

WWW: <http://www.williamspublishing.com>

---

# Python и финансовые вычисления

Это вводная часть, включающая всего две главы.

- **Глава 1** посвящена знакомству с языком программирования Python. Вы узнаете, почему именно он выбран основным инструментом разработки приложений, предназначенных для финансовых расчетов и анализа финансовых данных.
- **Глава 2** посвящена инфраструктуре Python. Вы узнаете о том, как начать работу в среде Python для выполнения финансовых расчетов в интерактивном режиме.



---

# Python как инструмент финансовых расчетов

Банки — самые высокотехнологичные предприятия.

*Хуго Банцигер*

## Язык программирования Python

Python — это универсальный язык программирования высокого уровня, который широко применяется в самых разных областях. На официальном сайте Python (<https://www.python.org/doc/essays/blurb>) он характеризуется следующим образом.

Python — это интерпретируемый, объектно-ориентированный, высокоуровневый язык программирования с динамической семантикой. Высокоуровневые структуры данных в сочетании с динамической типизацией и динамическим связыванием делают его удобным инструментом быстрой разработки приложений, а также привлекательным средством написания сценариев и языком интеграции существующих компонентов. Простой в изучении и понятный синтаксис языка ориентирован на удобство чтения кода, что позволяет снизить затраты на сопровождение программ. В Python поддерживаются модули и пакеты, что облегчает повторное использования кода и способствует повышению модульности программ. Интерпретатор и богатая стандартная библиотека языка свободно доступны как в виде исходных кодов, так и в бинарном виде для большинства компьютерных платформ и могут свободно распространяться.

Приведенное описание как нельзя лучше показывает, почему Python стал одним из самых популярных языков программирования. В наши дни им пользуются как начинающие программисты, так и профессиональные разработчики. Он применяется везде: в школах, университетах, крупных компаниях и финансовых учреждениях, а также на веб-сайтах и в научной среде.

Среди преимуществ Python можно выделить следующее.

#### *Открытый исходный код*

Большинство программных инструментов и библиотек Python находится в свободном доступе и распространяется на условиях открытой лицензии.

#### *Интерпретируемость*

Эталонной реализацией считается интерпретатор CPython, преобразующий код Python в байтовый код, непосредственно выполняемый компьютером.

#### *Мультипарадигменность*

Python поддерживает самые разные стили программирования, в частности объектно-ориентированный, императивный, функциональный и процедурный.

#### *Универсальность*

Python подходит для написания как простых интерактивных приложений, так и сложных многофункциональных программ. При этом он прекрасно справляется с выполнением как низкоуровневых системных операций, так и высокоуровневых аналитических задач.

#### *Многоплатформенность*

Python доступен для всех популярных операционных систем, таких как Windows, Linux и macOS. Он позволяет писать как настольные, так и веб-приложения, которые можно запускать как в огромных серверных кластерах, так и на небольших устройствах типа Raspberry Pi.

#### *Динамическая типизация*

В Python типы данных определяются непосредственно на этапе выполнения, а не объявляются статически, как в большинстве компилируемых языков программирования.

#### *Поддержка отступов в коде*

В отличие от других языков программирования в Python структура программы (группировка операторов) задается отступами, что позволяет не использовать фигурные скобки, точки с запятой и другие разделители.

#### *Сбор мусора*

В Python реализован автоматический механизм сбора мусора, что избавляет программиста от необходимости заниматься управлением памятью.

Точнее всего философия языка Python изложена в следующих 20 положениях, известных как “The Zen of Python”. Для ознакомления с ними достаточно выполнить команду `import this` в любой интерактивной оболочке Python.

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious
    way to do it.
Although that way may not be obvious at first unless
    you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be
    a good idea.
Namespaces are one honking great idea -- let's do more
    of those!
```

В переводе на русский язык они звучат так.

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное лучше, чем плотное.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.



- При этом практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Если не замалчиваются явно.
- Встретив двусмысленность, отбрось искушение угадать.
- Должен существовать один — и желательно только один — очевидный способ сделать это.
- Хотя он поначалу может быть и не очевиден, если вы не голландец.
- Сейчас лучше, чем никогда.
- Хотя никогда зачастую лучше, чем прямо сейчас.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.
- Пространства имен — отличная вещь! Давайте будем делать их больше!

## Краткая история Python

Некоторые до сих пор считают Python новым языком, хотя он существует уже достаточно давно. Его разработка началась в 80-х годах XX века программистом из Нидерландов Гвидо ван Россумом. Он длительное время участвовал в развитии языка на правах *великодушного пожизненного диктатора* (Benevolent Dictator For Life — BDFL) — почетный титул, который ему присвоило сообщество Python. Только в июле 2018 года, после нескольких десятилетий активной работы, Гвидо ван Россум принял решение оставить свой почетный пост BDFL и стать рядовым разработчиком.

Ниже перечислены основные вехи развития Python:

- Python 0.9.0, 1991 год (первый релиз);
- Python 1.0, 1994 год;
- Python 2.0, 2000 год;
- Python 2.6, 2008 год;
- Python 3.0, 2008 год;
- Python 3.1, 2009 год;
- Python 2.7, 2010 год;
- Python 3.2, 2011 год;
- Python 3.3, 2012 год;

- Python 3.4, 2014 год;
- Python 3.5, 2015 год;
- Python 3.6, 2016 год;
- Python 3.7, 2018 год;
- Python 3.8, 2019 год;

Новичков часто сбивает с толку тот факт, что начиная с 2008 года актуальными остаются сразу две версии Python, которые существуют параллельно. Это связано с наличием огромного количества программ, написанных на Python 2.6/2.7, которые продолжают активно использоваться. В то же время новые проекты пишутся на Python 3.6–3.8. Примеры, приводимые в книге, ориентированы на версию 3.7.

## Экосистема Python

Python — это не только язык программирования, но и целая экосистема, включающая большое количество библиотек и различных программных инструментов. Их необходимо *импортировать* по мере необходимости (как, например, библиотеку построения диаграмм) или запускать в виде отдельного системного процесса (как, например, интерактивную среду разработки). Операция импорта делает пакет доступным в текущем пространстве имен и в текущем процессе интерпретатора.

В базовый дистрибутив Python уже входит большой набор пакетов и модулей, расширяющих функциональность интерпретатора. Этот набор называется *стандартная библиотека Python* (<https://docs.python.org/3/library/index.html>). Например, общие математические вычисления не требуют подключения каких-либо модулей, тогда как специализированные математические функции импортируются через модуль `math`.

```
In [2]: 100 * 2.5 + 50
Out[2]: 300.0
```

```
In [3]: log(1) ❶
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-74f22a2fd43b> in <module>
----> 1 log(1) ❶

NameError: name 'log' is not defined
```

```
In [4]: import math ❷
```

```
In [5]: math.log(1) ❷
```

```
Out[5]: 0.0
```

- ❶ Без импорта соответствующего модуля эта строка вызовет ошибку.
- ❷ После импорта модуля `math` математические вычисления выполняются корректно.

Модуль `math` — это стандартный компонент, доступный в любом дистрибутиве Python, но есть множество других пакетов, которые устанавливаются опционально, после чего применяются так же, как и стандартные модули. Такие пакеты доступны на специализированных сайтах в Интернете. Для работы с ними рекомендуется использовать менеджер пакетов (он будет рассмотрен в главе 2), который гарантирует совместимость всех библиотек.

Приведенные примеры ориентированы на две самые популярные среды: IPython (<http://ipython.org/>) и Jupyter (<https://jupyter.org/>). Проект IPython задумывался как улучшенная интерактивная оболочка Python, но со временем превратился в *интегрированную среду разработки* (Integrated Development Environment — IDE) с поддержкой таких средств, как профилирование и отладка. Расширенные инструменты редактирования кода доступны через внешние редакторы типа Vim (<http://vim.org>), которые можно интегрировать в IPython.

IPython существенно расширяет возможности стандартной интерактивной оболочки. Среди ключевых дополнений — улучшенные средства работы с историей командной строки и поддержка инспектирования объектов. Например, чтобы просмотреть справку по функции (`docstring`), достаточно предварить или завершить ее имя знаком вопроса (добавление двух знаков вопроса позволит получить расширенную справку).

Первоначально IPython поставлялся в двух вариантах: как оболочка и как браузерная версия (*Notebook*). Последняя приобрела настолько большую популярность, что была выделена в отдельный многоязыковой проект, получивший название Jupyter. Как следствие, среда Jupyter Notebook наследует все самое лучшее от IPython, дополняя ее множеством расширенных инструментов, особенно в части визуализации данных.

## Круг пользователей Python

Неправильно полагать, будто Python интересен только профессиональным разработчикам. Это универсальный язык, применяемый в самых разных областях, в том числе для научных расчетов.

*Профессиональные разработчики* находят в Python все то, что необходимо для создания крупномасштабных приложений. В Python поддерживаются практически все основные парадигмы программирования и имеются средства для решения любых задач. Профессионалы часто пишут собственные библиотеки и классы и по максимуму задействуют весь стек научных пакетов.

*Разработчики специализированных и научных приложений*, как правило, активно пользуются конкретными пакетами и библиотеками, адаптируют экосистему под собственные задачи и постоянно дорабатывают и оптимизируют создаваемые программы. Они подолгу экспериментируют в рамках интерактивных сеансов, исследуя и визуализируя имеющиеся наборы данных.

*Рядовые программисты* обращаются к Python для решения конкретных задач, когда заранее известно, что имеются готовые наработки. Например, программист может посетить страницу с галереей визуализаций библиотеки `matplotlib`, скопировать оттуда готовый фрагмент кода и вставить его в собственный проект, адаптировав к условиям задачи. Это сильно ускоряет процесс разработки.

Есть еще одна многочисленная категория пользователей Python: *начинающие программисты*. На сегодняшний день Python стал одним из самых популярных языков, с которого начинают изучать программирование в университетах, колледжах и даже школах. Причиной служит простой и понятный синтаксис языка, доступный даже для неспециалистов, а также поддержка самых разных стилей программирования.

## Стек научных пакетов

Существует целый набор специализированных пакетов, условно объединяемых в так называемый *научный стек*. Рассмотрим основные компоненты данного стека.

### NumPy

Библиотека, обеспечивающая поддержку многомерных массивов, предназначенных для хранения однородных и неоднородных данных. Включает оптимизированные функции/методы для работы с такими массивами.

## SciPy

Коллекция пакетов, реализующих функции для научных и финансовых расчетов. В частности, включает функции интерполяции кубических сплайнов и численного интегрирования.

## matplotlib

Один из самых популярных пакетов визуализации данных и построения как двумерных, так и трехмерных диаграмм на Python.

## pandas

Основанная на NumPy библиотека, предназначенная для обработки и анализа временных рядов и табличных данных. Тесно интегрирована с пакетами matplotlib (визуализация данных) и PyTables (хранение и загрузка данных).

## Scikit-learn

Популярный пакет машинного обучения, содержащий унифицированные реализации множества алгоритмов, таких как классификация и кластеризация.

## PyTables

Популярная надстройка для работы с файлами формата HDF5. Реализует оптимизированные операции дискового ввода-вывода.

В зависимости от конкретной задачи могут потребоваться и другие пакеты, которые зачастую реализуются поверх одного из вышеперечисленных базовых пакетов. Но в целом ключевыми структурными элементами выступают класс `ndarray` из пакета NumPy (глава 4) и класс `DataFrame` из пакета pandas (глава 5).

Если рассматривать Python только как язык программирования, то необходимо сказать, что есть и другие языки, которые могут составить ему конкуренцию в плане синтаксиса и эффективности, например Ruby. На официальном сайте ([www.ruby-lang.org](http://www.ruby-lang.org)) он характеризуется следующим образом.

Динамический язык программирования с открытым исходным кодом, для которого характерны простота и высокая производительность. Обладает элегантным синтаксисом, благодаря которому код легко читать и так же легко писать.

Те, кто программировали на Python, наверняка согласятся с тем, что аналогичным образом можно охарактеризовать и этот язык. Но что отличает Python от конкурентов наподобие Ruby, так это наличие научного стека, благодаря которому можно отказаться от специализированных языков и пакетов

типа Matlab и R. Кроме того, в языке по умолчанию имеется все необходимое, к примеру, для опытного веб-разработчика или системного администратора. К тому же Python отлично взаимодействует со специализированными языками типа R, поэтому вопрос обычно не в том, выбирать Python или нет, а в том, какой из языков разработки будет основным в конкретном проекте.

## Технологии в финансовой отрасли

Получив общее представление о возможностях Python, поговорим о роли современных технологий в финансовом анализе. Нас в первую очередь будут интересовать то, какое влияние Python оказывает и, что самое важное, будет оказывать на развитие финансовой индустрии.

По сути, в применении передовых технологий в финансовой отрасли нет ничего необычного. Однако недавние инновации, а также законодательные изменения привели к тому, что банки и различные финансовые учреждения, например хедж-фонды, все больше стали превращаться из финансовых посредников в технологические компании. Технологическая база начала рассматриваться как один из ключевых активов для большинства финансовых компаний по всему миру, что дает как преимущества, так и недостатки. Следует разобраться, почему так происходит.

## Инвестиции в технологии

Банки и другие финансовые учреждения образуют целую индустрию, которая ежегодно тратит огромные средства на передовые технологии. Следующая цитата показывает не только важность технологий для финансовой отрасли, но и важность самой этой отрасли для технологического сектора.

Согласно аналитическим данным, собранным компанией IDC (International Data Corporation), к 2021 году общемировые затраты финансового сектора на информационные технологии вырастут до 500 млрд долл. по сравнению с 440 млрд долл. в 2018 году.

IDC (июнь 2018 г.)

Банки и финансовые учреждения уже давно втянуты в гонку, направленную на переход к цифровой модели управления.

Согласно прогнозам в 2017 году банковские инвестиции в передовые технологии в Северной Америке должны составить около 19,9 млрд долл.

Банки разрабатывают современные информационные системы и новые технологические решения с целью получения конкурентных преимуществ на глобальном рынке и привлечения клиентов, ориентированных на использование онлайн-служб и мобильных приложений. Для глобальных финансово-технологических компаний это прекрасная возможность предложить новые идеи и программные решения банковской индустрии.

Statista

Транснациональные банки задействуют тысячи разработчиков для обслуживания существующих систем и создания новых. Крупные инвестиционные компании, сталкиваясь с ростом технологических требований, зачастую имеют бюджет ИТ-направления в несколько миллиардов долларов.

## Технологии как движущая сила

Развитие технологий способствует внедрению инноваций и повышению эффективности финансового сектора. Как правило, проекты в данной сфере следуют концепции полного перехода на цифровые модели.

За последние несколько лет финансовая индустрия претерпела радикальные технологические изменения. Многие руководители требуют от своих ИТ-отделов повышать эффективность и внедрять качественно новые, инновационные услуги, при этом поддерживая существующие системы, но снижая затраты на их эксплуатацию. А тем временем на рынок все активнее выходят финтех-стартапы, предлагая пользовательские системы, разработанные с нуля, а не основанные на унаследованных решениях.

PwC 19th Annual Global CEO Survey, 2016<sup>1</sup>

Побочным эффектом повышения эффективности становится то, что конкурентные преимущества приходится искать во все более сложных решениях. Это неизбежно влечет за собой рост рисков и затрат, связанных с соблюдением регуляторных требований. Финансовый кризис 2007–2008 годов показал, какую опасность могут нести новые технологии. С аналогичными технологическими рисками сталкивается и финансовый сектор, что нагляднее всего проявилось в обвале фондового рынка, который имел место в мае 2010 года, когда вследствие применения автоматизированных биржевых систем обрушение индексов достигло триллионного масштаба. (Вопросы, связанные с алгоритмической торговлей финансовыми инструментами, будут рассматриваться в части IV.)

---

<sup>1</sup> <https://pwc.to/10YT02d>.

## Технологии и кадры решают все

Как видим, с одной стороны, развитие технологий ведет к снижению затрат, а с другой — только постоянные инвестиции в разработку и внедрение новых решений позволяют финансовым компаниям добиваться конкурентных преимуществ и укреплять свои позиции на рынке. Во многих областях крупномасштабные инвестиции в технологии и квалифицированный персонал служат ключом к выживанию на рынке. В качестве примера рассмотрим рынок анализа деривативов.

В среднем, если рассматривать весь цикл существования программного обеспечения, фирмы, внедряющие собственные стратегии внебиржевого управления деривативами, должны вложить от 25 до 36 млн долларов только на создание, поддержку и улучшение соответствующей программной библиотеки.

Дин [1]<sup>2</sup>

Построение полнофункциональной библиотеки анализа деривативов — не только трудоемкий и затратный процесс, для него еще и требуется наличие *достаточного числа экспертов*. К тому же сами эксперты должны располагать всеми необходимыми технологиями и программными средствами. В этом отношении современная экосистема Python выглядит очень привлекательно, снабжая разработчиков намного более эффективными и дешевыми инструментами, чем, к примеру, 10 лет назад. (Тема анализа деривативов рассматривается в части V, где мы напишем небольшую, но достаточно эффективную и гибкую библиотеку, задействуя только стандартные пакеты Python.)

Следующая цитата возвращает нас в эпоху LTCM — одного из крупнейших хедж-фондов, история которого завершилась крахом в конце 1990-х.

Мериуэзер вложил 20 млн долл. в разработку передовой компьютерной системы и нанял команду первоклассных финансовых инженеров для управления компанией LTCM, офис которой находился в Гринвиче, штат Коннектикут. Это была система управления рисками промышленного масштаба.

Паттерсон [3]

Те вычислительные мощности, на которые Мериуэзер потратил миллионы долларов, сегодня можно купить всего за несколько тысяч долларов или поступить еще проще, арендовав их в одной из облачных служб в рамках гибкого плана. (О развертывании облачной инфраструктуры для интерактивного финансового анализа и разработки приложений на Python будет рассказываться в главе 2.) Бюджеты подобной профессиональной инфраструктуры составляют

---

<sup>2</sup> См. раздел “Дополнительные ресурсы” в конце главы.



от нескольких долларов в месяц. В то же время сами системы биржевой торговли, финансового анализа и управления рисками стали настолько сложными, что современным компаниям приходится внедрять IT-инфраструктуры, насчитывающие десятки тысяч вычислительных ядер.

## В погоне за скоростью, производительностью и объемами данных

Если говорить о тех аспектах финансовых вычислений, на которые технологические изменения оказывают наибольшее влияние, то это скорость и интенсивность финансовых транзакций. Льюис [2] описывает понятие *флеш-трейдинга* — процесса высокочастотной биржевой торговли, когда сделки совершаются с максимально возможной скоростью.

С одной стороны, повышение доступности биржевых данных требует от финансовых компаний принятия решений в реальном времени. А с другой — внедрение высокочастотной торговли приводит к еще большему росту объемов данных. Оба процесса идут рука об руку, непрерывно снижая временной масштаб финансовых транзакций. Впервые эта тенденция была выявлена еще десять лет назад.

В 2008 году фонд Medallion, основанный компанией Renaissance Technologies, получил небывалую маржу в размере 80%, сумев воспользоваться крайней волатильностью рынка благодаря своим сверхбыстрым компьютерам. Крупнейшим бенефициарием в тот год стал Джим Симонс, который положил в карман солидные 2,5 млрд долл.

Паттерсон [3]

Данные о биржевых котировках одного индекса на конец торгов, собранные за 30 лет, содержатся примерно в 7500 записях. Именно на этих данных базируются все современные финансовые теории, в частности *портфельная теория Марковица* (Mean-variance Portfolio Theory — MPT), *модель ценообразования капитальных активов* (Capital Asset Pricing Model — CAPM) и *стоимость под риском* (value-at-risk — VaR).

Для сравнения: в обычный торговый день курс акций компании Apple (AAPL) в среднем изменяется около 15 тыс. раз в час, что в два раза больше, чем количество котировок, собранных за последних 30 лет. Анализ таких объемов данных сопряжен с целым рядом трудностей.

### Обработка данных

Сегодня уже недостаточно обрабатывать только сведения о котировках акций и других финансовых инструментов на момент закрытия торгов.

Слишком много событий происходит в течение каждого торгового дня, а для некоторых инструментов — 24 часа в сутки 7 дней в неделю.

### *Скорость анализа данных*

Биржевые решения должны приниматься за миллисекунды или даже быстрее, а это означает, что нужны соответствующие аналитические системы, способные обрабатывать огромные объемы данных в реальном времени.

### *Теоретическая база*

Несмотря на то что традиционные финансовые теории далеки от идеала, они прошли испытание временем. Но когда речь идет о принятии решений в милли- и микросекундных интервалах, надежных теорий пока еще недостаточно.

Указанные трудности можно преодолеть только с помощью современных технологий. Самое удивительное то, что даже отсутствие надежных финансовых теорий часто компенсируется технологическими преимуществами, поскольку алгоритмы высокочастотной торговли задействуют микроструктурные биржевые элементы (поток заявок, спреда и т.п.), а не опираются на теоретические суждения.

## **Анализ в реальном времени**

За последние годы в финансовой сфере резко возросла важность *финансового анализа*. Это прямое следствие повышения скорости и производительности вычислений, а также увеличения объемов данных. По сути, реакцией рынка стало появление анализа в реальном времени.

Под финансовым анализом мы понимаем применение современных технологий, программных решений и алгоритмов для получения аналитических данных или принятия решений. В качестве примера можно привести оценку того, какое влияние на продажи окажет изменение стоимости банковского продукта, или крупномасштабную корректировку кредитной оценки (Credit Valuation Adjustment — CVA) для сложных портфелей деривативов крупного инвестиционного банка.

В этом контексте финансовые учреждения сталкиваются с двумя основными трудностями.

### *Большие данные*

Банкам и другим финансовым компаниям приходилось иметь дело с огромными массивами данных еще до начала эпохи “больших данных”.

Как бы там ни было, сегодня объемы анализируемых данных настолько велики, что требуют постоянного увеличения как вычислительной мощности компьютерных систем, так и размера хранилищ данных.

### *Расчеты в реальном времени*

В прошлом инвестиционные решения принимались на основе тщательно спланированных стратегий и процессов управления рисками. В нынешнюю эпоху решения приходится принимать в реальном времени. Задачи, которые раньше решались в офисе на протяжении рабочего дня, сегодня решаются прямо на бирже в процессе торгов.

Здесь снова прослеживается связь между развитием технологий и практикой ведения бизнеса. С одной стороны, существует постоянная потребность в повышении скорости и точности аналитических решений за счет внедрения современных технологий. А с другой стороны, появление новых технологий делает возможными новые аналитические подходы, которые всего несколько лет назад считались невозможными (или нереалистичными исходя из бюджетных соображений).

Одной из ключевых тенденций в сфере финансового анализа стало внедрение параллельных архитектур многоядерных вычислений на стороне центрального процессора и массово-параллельных архитектур на основе графических процессоров. Современные графические процессоры содержат тысячи вычислительных ядер, что заставляет переосмысливать алгоритмы параллельных вычислений. Но для того чтобы извлечь преимущества из новых аппаратных платформ, пользователям приходится осваивать новые языки и парадигмы программирования.

## **Python для финансовых расчетов**

В предыдущем разделе рассматривались основные аспекты применения современных технологий в финансовой среде:

- стоимость технологий для финансовой индустрии;
- технологии как движущая сила бизнеса и инноваций;
- технологии и квалифицированные кадры как барьеры для выхода на финансовый рынок;
- увеличение скорости и интенсивности вычислений, а также объемов данных;
- необходимость анализа данных в реальном времени.

В этом разделе вы узнаете о том, как Python помогает преодолевать возникающие проблемы. Но для начала следует познакомиться с общезычковыми инструментами и синтаксисом Python.

## Синтаксис Python, применяемый в финансовых вычислениях

Большинство разработчиков, которые только начинают знакомиться с инструментами Python, применяемыми в финансовых расчетах, неизбежно сталкиваются с алгоритмическими проблемами. То же самое будет испытывать ученый, которому нужно решить дифференциальное уравнение, вычислить интеграл или просто визуализировать данные. На этом этапе еще никто не думает о формальном процессе разработки, тестировании, документации или развертывании готового приложения. И именно здесь люди начинают ценить Python как язык программирования. Причина заключается в том, что синтаксис Python в достаточной степени приближен к математическому синтаксису, применяемому для решения научных и финансовых задач.

В качестве иллюстрации можно взять любой финансовый алгоритм, например оценку европейского колл-опциона методом Монте-Карло. Предположим, применяется модель Блэка — Шоулза — Мертона (Black–Scholes–Merton — BSM), в которой рискoвость опциона описывается через геометрическое броуновское движение.

Примем следующие значения параметров:

- начальный уровень индекса,  $S_0 = 100$ ;
- страйк-цена опциона,  $K = 105$ ;
- срок исполнения опциона,  $T = 1$  год;
- безрисковая краткосрочная ставка,  $r = 0,05$ ;
- волатильность доходности актива,  $\sigma = 0,2$ .

В модели Блэка — Шоулза — Мертона уровень индекса при экспирации опциона представляется случайной величиной, рассчитываемой по следующей формуле, в которой случайная переменная  $z$  имеет нормальное распределение вероятностей (уравнение 1.1).

*Уравнение 1.1. Уровень индекса при экспирации опциона в модели Блэка — Шоулза — Мертона*

$$S_T = S_0 \exp \left( \left( r - \frac{1}{2} \sigma^2 \right) T + \sigma \sqrt{T} z \right).$$

Ниже приведено алгоритмическое описание процедуры оценки опциона по методу Монте-Карло.

1. Образуйте множество  $I$  псевдослучайных чисел  $z(i)$ ,  $i \in \{1, 2, \dots, I\}$  из стандартного нормального распределения.
2. Вычислите уровень индекса при экспирации опциона  $S_T(i)$  для всех заданных  $z(i)$ , воспользовавшись формулой из уравнения 1.1.
3. Вычислите внутренние стоимости по следующему уравнению:

$$h_T(i) = \max(S_T(i) - K, 0).$$

4. Определите текущую стоимость опциона по методу Монте-Карло (уравнение 1.2).

*Уравнение 1.2. Расчет стоимости европейского опциона по методу Монте-Карло*

$$C_0 \approx e^{-rT} \frac{1}{I} \sum_I h_T(i).$$

Следующим этапом будет перевод алгоритма на язык Python. Решение выглядит следующим образом.

```
In [6]: import math
import numpy as np ❶

In [7]: S0 = 100. ❷
K = 105. ❷
T = 1.0 ❷
r = 0.05 ❷
sigma = 0.2 ❷

In [8]: I = 100000 ❷

In [9]: np.random.seed(1000) ❸

In [10]: z = np.random.standard_normal(I) ❹

In [11]: ST = S0 * np.exp((r - sigma ** 2 / 2) * T + sigma *
math.sqrt(T) * z) ❺

In [12]: hT = np.maximum(ST - K, 0) ❻
```

```
In [13]: C0 = math.exp(-r * T) * np.mean(hT) ⑦
```

```
In [14]: print('Стоимость европейского колл-опциона:  
          {0:5.3f}'.format(C0)) ⑧  
Стоимость европейского колл-опциона: 8.019.
```

- ① Финансовые расчеты в данном случае выполняются средствами пакета NumPy.
- ② Задаем параметры модели.
- ③ Фиксируем затравочное значение для генератора случайных чисел.
- ④ Формируем множество случайных чисел со стандартным распределением вероятности.
- ⑤ Вычисляем цену актива при экспирации опциона.
- ⑥ Вычисляем премию по опциону.
- ⑦ Оценка опциона по методу Монте-Карло.
- ⑧ Выводим результат оценки на экран.

Рассмотрим ключевые моменты.

#### *Понятный синтаксис*

Синтаксис Python действительно схож с математическим языком, что особенно заметно при задании параметров модели.

#### *Трансляция задачи*

Каждое математическое уравнение и каждый шаг алгоритма представляется в коде Python *одной* инструкцией.

#### *Векторизация*

Одно из достоинств пакета NumPy — компактный векторизованный синтаксис, что позволяет выполнить, к примеру, 100 000 вычислений с помощью одной команды.

Приведенный выше код можно легко выполнить в интерактивной среде, такой как IPython или Jupyter Notebook. Но если код планируется выполнять на регулярной основе, его следует сохранить в виде *модуля* (сценария), представляющего собой отдельный файл Python (с технической точки зрения простой текстовый документ) с расширением *.py*. В данном случае модуль можно сохранить в файле *bsm\_mcs\_euro.py* (листинг 1.1).

### Листинг 1.1. Оценка европейского колл-опциона по методу Монте-Карло

```
#
# Оценка европейского колл-опциона
# (модель Блэка – Шоулза – Мертона)
# bsm_mcs_euro.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import math
import numpy as np

# Значения параметров
S0 = 100.    # начальный уровень индекса
K = 105.     # страйк-цена опциона
T = 1.0      # срок исполнения опциона
r = 0.05     # безрисковая краткосрочная ставка
sigma = 0.2  # волатильность

I = 100000   # количество этапов моделирования

# Алгоритм оценки
z = np.random.standard_normal(I) # генерация псевдослучайных чисел

# Цена актива при экспирации опциона
ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma *
                 math.sqrt(T) * z)
hT = np.maximum(ST - K, 0)        # премия по опциону
C0 = math.exp(-r * T) * np.mean(hT) # оценка по методу Монте-Карло

# Вывод результата
print('Стоимость европейского колл-опциона: %5.3f.' % C0)
```

Данный алгоритмический пример наглядно демонстрирует, насколько удобен синтаксис Python для выполнения научных расчетов. Он позволяет легко перейти от постановки задачи и ее математического описания к программной реализации. Порядок действий выглядит так.

- **Обычный язык.** Позволяет в устном и письменном виде выразить суть научной или финансовой задачи.
- **Математический язык.** Позволяет максимально четко и лаконично описать и смоделировать абстрактные аспекты задачи, алгоритм ее решения, числовые параметры и т.п.

- **Python.** Позволяет *технически моделировать и реализовать* абстрактные аспекты задачи, алгоритм ее решения, числовые параметры и т.п.



### Математические аспекты синтаксиса Python

Вряд ли вам удастся найти другой язык программирования, настолько близкий синтаксически к языку математики, как Python. В большинстве случаев математические выражения и формулы транслируются в Python почти в неизменном виде, что делает его невероятно эффективным языком разработки и написания приложений для финансовых расчетов.

Нередко в применяемый стек языков добавляется четвертый компонент: *псевдокод*. Его роль заключается в представлении математических формул, лежащих в основе финансовых расчетов, в наиболее понятном для программной реализации виде. Помимо самого алгоритма, псевдокод должен учитывать особенности обработки данных компьютером.

Практика применения псевдокода возникла вследствие огромного различия между формальным математическим представлением задачи и его описанием в большинстве компилируемых языков программирования. Очень часто написанный на таких языках код настолько усложняется добавлением обязательных технических элементов, что его затруднительно сопоставить с исходными математическими выкладками.

К счастью, при работе с Python отдельный этап составления псевдокода не нужен, поскольку синтаксис языка почти аналогичен языку математики, а чисто технические дополнения сведены к минимуму. Для этого в Python предусмотрено множество высокоуровневых конструкций. Конечно, у подобного подхода есть и недостатки, но в целом можно сказать, что всякий раз, когда возникает такая необходимость, в Python можно следовать строгим процедурам кодирования, как и в любых других языках программирования. В этом смысле Python берет все лучшее из двух миров: *высокоуровневую абстракцию и строгую реализацию*.

## Эффективность и производительность кода Python

Если рассматривать ситуацию на высоком уровне, то преимущества Python можно оценить по следующим аспектам.

### Эффективность

Поможет ли Python получить результаты быстрее и дешевле?



## Производительность

Позволит ли Python достичь большего при тех же самых ресурсах (люди, средства и т.п.)?

## Качество

Что такого позволяет делать Python, чего не могут дать другие технологии?

Навряд ли на такие вопросы можно дать исчерпывающие ответы, но все же приведем ряд аргументов.

## Высокая скорость вычислений

Область, в которой эффективность Python становится очевидной, — это интерактивный анализ данных. Именно здесь проявляются преимущества таких мощных инструментов, как IPython, Jupyter Notebook и программных пакетов наподобие pandas.

Рассмотрим студента экономического факультета, который пишет диплом, исследуя индекс S&P 500. Ему необходимо проанализировать значения индекса за несколько лет и попытаться найти доказательства того, что его волатильность, в противоположность принятым во многих финансовых моделях предположениям, изменяется со временем, а потому ее нельзя выражать константой. Результаты следует представить в графическом виде. Для решения такой задачи нужно выполнить следующие действия:

- загрузить данные о значениях индекса S&P 500 из Интернета;
- вычислить среднеквадратическое отклонение логарифмической доходности (волатильности) в годовом разрезе;
- построить графики изменения индекса S&P 500 и волатильности.

Еще совсем недавно такого рода задачи считались настолько сложными, что их решение было под силу только профессиональным финансовым аналитикам. Сегодня же с ней легко справится даже студент экономического факультета. Ниже приведен соответствующий код (на синтаксис можете пока не обращать внимания — мы все детально рассмотрим в последующих главах).

```
In [16]: import numpy as np ❶  
         import pandas as pd ❶  
         from pylab import plt, mpl ❷  
  
In [17]: plt.style.use('seaborn') ❷  
         mpl.rcParams['font.family'] = 'serif' ❷  
         %matplotlib inline
```

```

In [18]: data = pd.read_csv('.././source/tr_eikon_eod_data.csv',
                             index_col=0, parse_dates=True) ❸
data = pd.DataFrame(data['.SPX']) ❹
data.dropna(inplace=True) ❺
data.info() ❻
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
Data columns (total 1 columns):
.SPX      2138 non-null float64
dtypes: float64(1)
memory usage: 33.4 KB

In [19]: data['Доходность'] = np.log(data / data.shift(1)) ❼
data['Волатильность'] =
    data['Доходность'].rolling(252).std() * np.sqrt(252) ❽

In [20]: data[['.SPX', 'Волатильность']].plot(subplots=True,
                                                figsize=(10, 6)); ❾

```

- ❶ Импорт пакетов NumPy и pandas.
- ❷ Импорт библиотеки matplotlib и настройка стиля диаграмм, отображаемых в среде Jupyter.
- ❸ Метод `pd.read_csv()` позволяет загрузить данные из CSV-файла, хранящегося локально или на сервере.
- ❹ Формирование подмножества данных и исключение нечисловых значений (NaN).
- ❺ Вывод метаданных о выбранном наборе данных.
- ❻ Вычисление логарифмической доходности в векторизованной форме (без применения циклов).
- ❼ Определение среднегодовой волатильности.
- ❽ Визуализация двух временных рядов.

Результат данного интерактивного сеанса представлен на рис. 1.1. Просто поразительно, как с помощью всего нескольких строк кода нам удалось решить три сложные задачи финансового анализа: сбор данных, выполнение повторяющихся математических вычислений и визуализация результатов. Обратите внимание на то, что благодаря использованию пакета `pandas` с

временными рядами можно работать почти так же легко, как и с вещественными числами.



Рис. 1.1. Уровни индекса S&P 500 и среднегодовая волатильность

Как показывает данный пример, при правильном выборе инструментов и пакетов Python, предлагающих соответствующие высокоуровневые абстракции, финансовый аналитик получает возможность сконцентрироваться на решении конкретной профессиональной задачи, а не на технических аспектах программной реализации. Это позволяет аналитику быстрее реагировать на изменения рыночных условий, делая оценки практически в режиме реального времени и тем самым опережая своих конкурентов. Таким образом, *увеличение эффективности* может легко давать коммерческий эффект.

## Повышение производительности

Как правило, критики признают, что Python обладает понятным синтаксисом, позволяющим писать эффективный код. Однако по своей природе это интерпретируемый язык, а потому существует предубеждение, будто бы код Python слишком медленный для выполнения трудоемких финансовых расчетов. В определенных ситуациях это действительно так, но в реальности все зависит от выбранного подхода. Никто не мешает вам писать эффективные приложения! Можно выделить три стратегии повышения производительности.

## Идиомы и парадигмы программирования

Python позволяет решать одни и те же задачи разными способами, которые характеризуются разной производительностью. Выбрав правильный способ реализации (т.е. разумно используя структуры данных, избегая циклов в рамках векторизации и применяя пакеты наподобие `pandas`), можно существенно повысить скорость любых финансовых вычислений.

### Компиляция кода

Существуют специальные пакеты повышения производительности, содержащие скомпилированные версии важных функций или выполняющие статическую либо динамическую компиляцию кода Python, что на порядок ускоряет выполнение важных функций в сравнении с чистым кодом Python. Самые популярные пакеты такого рода — `Cython` и `Numba`.

### Параллельные вычисления

Многие вычислительные процессы, особенно в финансовой области, можно существенно ускорить за счет параллельного выполнения. В Python это можно сделать очень легко.



#### Высокопроизводительные вычисления на Python

Сам по себе Python нельзя назвать высокопроизводительным языком. Но он превратился в идеальную платформу для доступа к производительным технологиям, что делает его своего рода связующим языком.

Рассмотрим простой, но реалистичный пример, в котором задействуются все три вышеуказанные стратегии (подробнее они рассматриваются в последующих главах). В финансовом анализе часто возникает задача оценки сложных математических выражений применительно к большим числовым массивам. В Python для этого есть собственные инструменты.

```
In [21]: import math
         loops = 2500000
         a = range(1, loops)
         def f(x):
             return 3 * math.log(x) + math.cos(x) ** 2
         %timeit r = [f(x) for x in a]
         1.59 s ± 41.2 ms per loop (mean ± std. dev. of 7 runs,
                               1 loop each)
```

Как видите, на выполнение функции `f()` 2,5 млн раз интерпретатору Python потребовалось примерно 1,6 с. Ту же самую задачу можно решить с помощью *оптимизированных* (т.е. предварительно откомпилированных) функций пакета NumPy.

```
In [22]: import numpy as np
         a = np.arange(1, loops)
         %timeit r = 3 * np.log(a) + np.cos(a) ** 2
         87.9 ms ± 1.73 ms per loop (mean ± std. dev. of 7 runs,
                               10 loops each)
```

В результате время выполнения кода сократилось до 88 мс. Впрочем, в Python есть специальный программный пакет для решения таких задач: `numexpr`. Он *компилирует* все выражение, чтобы добиться еще большей, чем позволяет NumPy, производительности. Это, в частности, достигается за счет предотвращения хранения в памяти множественных копий объектов `ndarray`.

```
In [23]: import numexpr as ne
         ne.set_num_threads(1)
         f = '3 * log(a) + cos(a) ** 2'
         %timeit r = ne.evaluate(f)
         50.6 ms ± 4.2 ms per loop (mean ± std. dev. of 7 runs,
                               10 loops each)
```

Данный подход позволяет сократить время обработки данных еще сильнее — до 50 мс. Наряду с этим пакет `numexpr` располагает средствами параллелизации вычислений, что позволяет задействовать множественные потоки центрального процессора.

```
In [24]: ne.set_num_threads(4)
         %timeit r = ne.evaluate(f)
         22.8 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs,
                               10 loops each)
```

Параллелизация дополнительно сокращает время выполнения кода до 23 мс за счет применения четырех потоков. Таким образом, проведенная нами оптимизация позволила повысить производительность более чем в 90 раз. Причем это стало возможным без изменения базового алгоритма и без знания алгоритмов компиляции или параллельных вычислений. Такого рода программные средства доступны на высоком уровне даже для пользователей, не являющихся экспертами в Python. Но, разумеется, необходимо знать, какие средства есть в вашем распоряжении.

Рассмотренный пример показывает, что Python располагает целым рядом средств, позволяющих извлечь максимум из имеющихся ресурсов. Тем самым

мы получаем *повышение производительности*. При параллельном подходе, в отличие от последовательного, за то же самое время можно выполнить в 6 раз больше вычислений. Для этого достаточно указать интерпретатору на необходимость использования всех потоков центрального процессора, а не только одного.

## От прототипа к готовому приложению

Эффективность интерактивного анализа и высокая скорость вычислений — два ключевых преимущества Python. Но есть и еще одно, не очевидное на первый взгляд преимущество, которое может иметь стратегическое значение с точки зрения финансовых вычислений. Речь идет о возможности использовать Python на протяжении *всего производственного цикла*, от создания прототипа до выпуска готового приложения.

Практика современных финансовых компаний по всему миру, когда речь идет о разработке финансовых приложений, зачастую подразумевает двухэтапный процесс. На первом этапе к работе подключаются специалисты по *финансовой математике* (“кванты”), отвечающие за разработку модели и технического прототипа. Они предпочитают инструменты типа Matlab и R, удобные для быстрой интерактивной разработки приложений. На данной стадии вопросы производительности, стабильности, развертывания инфраструктуры, контроля доступа и управления версиями не имеют принципиального значения. Речь идет лишь о получении принципиально работоспособной модели и прототипа, соответствующего ключевым требованиям алгоритма или итогового приложения.

Как только прототип готов, в игру вступают *разработчики* из IT-отдела, которые отвечают за перевод кода прототипа в надежный, управляемый и производительный *производственный код*. Как правило, на этом этапе происходит сдвиг парадигмы, поскольку требованиям развертывания программной инфраструктуры соответствуют компилируемые языки, такие как C++ или Java. Кроме того, применяется формальный процесс разработки с задействованием профессиональных инструментов, систем управления версиями и пр.

Описанный двухэтапный процесс имеет ряд принципиальных недостатков.

### *Низкая эффективность*

Код прототипа не подлежит повторному использованию, и алгоритмы придется реализовывать дважды, что приводит к дополнительным тратам времени и ресурсов. Кроме того, при переписывании кода прототипа могут возникать ошибки.

### Квалификация исполнителей

Специалисты из разных отделов обладают разными навыками и применяют разные языки для разработки одного и того же. Более того, они даже выражают свои мысли по-разному.

### Унаследованный код

Полученный код приходится обслуживать на разных языках и платформах.

Python позволяет *упростить* весь цикл разработки, поскольку один набор инструментов применяется от первых интерактивных попыток получить рабочий прототип и до финального выпуска надежного и эффективного производственного кода. Упрощается не только взаимодействие специалистов из разных департаментов, но и обучение персонала, поскольку на всех этапах разработки финансового приложения задействуется один язык программирования. Устраняется также неизбежная избыточность и неэффективность, связанная с применением разных технологий на разных этапах процесса разработки. В целом Python предлагает *согласованную технологическую среду*, пригодную для решения любых задач финансового анализа, разработки финансовых приложений и реализации алгоритмов.

## Финансовые расчеты на основе данных и искусственного интеллекта

Все те соображения касательно применения технологий в финансовой отрасли, которые были сформулированы в первом издании книги, вышедшем в 2014 году, справедливы и сейчас. Но в последние годы в финансовой сфере проявились две важнейшие тенденции, которые несут фундаментальные изменения.

### Финансовые системы, управляемые данными

Некоторые фундаментальные финансовые теории, например *портфельная теория Марковица* (Modern Portfolio Theory — MPT) и *модель ценообразования капитальных активов* (Capital Asset Pricing Model — CAPM), разрабатывались еще в середине XX века. Они по-прежнему остаются краеугольным камнем образовательных программ, по которым студенты изучают экономику, финансы, финансовую инженерию и деловое администрирование. И это в определенной степени удивительно, так как их доказательная база достаточно

скудная, если не сказать больше, а имеющиеся наблюдения часто вступают в противоречие с тем, что утверждает та или иная теория. С другой стороны, они обладают очевидной популярностью, поскольку близки к нашему пониманию того, как должны работать финансовые рынки. Плюс это все-таки красивые математические теории, построенные на достаточно логичных, хоть и слишком упрощенных, рассуждениях.

*Научный метод*, применяемый, к примеру, в физике, базируется на *данных*, собираемых в ходе наблюдений и экспериментов. Только после этого строятся *гипотезы и теории*, которые затем снова проверяются на данных. Если научный тест подтверждает предположения ученых, то гипотеза или теория должным образом формализуется в виде диссертации либо академической публикации. Если же тест не дал результатов, то гипотеза или теория отвергается, и ученые начинают искать новые объяснения, которые не противоречили бы имеющимся данным. Поскольку законы физики фундаментальны и не меняются, открытие того или иного закона приводит к тому, что он становится частью наших знаний об окружающем мире на столетия вперед.

Финансовые теории исторически шли вразрез с научным методом. Зачастую их разрабатывали “от печки” на основе упрощенных математических моделей с целью поиска элегантных ответов на ключевые вопросы финансового анализа. Одно из популярных допущений — предположение о нормальном распределении доходности финансовых инструментов и линейной зависимости между процентными ставками. Но поскольку такого рода вещи редко наблюдаются на финансовых рынках, не удивительно, что эмпирических доказательств красивых теорий зачастую нет. Многие финансовые теории и модели сначала были сформулированы и описаны в научных изданиях и только впоследствии были проверены на реальных данных. В определенной степени это связано с тем, что в 1950–1970-е годы (и даже в следующие десятилетия) исследователи не располагали тем объемом финансовых данных, который сегодня есть даже у студентов-бакалавров.

Ситуация с доступностью финансовых данных радикально изменилась в середине 1990-х годов, и сегодня любой исследователь может получить доступ к огромным объемам исторических данных, а также к биржевым данным в реальном времени через потоковые службы. Это дает нам возможность вернуться к научному методу и начать с исследования данных, что всегда должно предшествовать появлению идей, гипотез, моделей и стратегий.

Следующий простой пример показывает, насколько легко в наши дни получить любые финансовые данные за интересующий период времени, используя только Python и подписку на программный интерфейс Eikon (<https://>



developers.refinitiv.com/eikon-apis/eikon-data-apis[]). Приведенный ниже код получает сведения о колебаниях курса акций компании Apple в течение одного часа дневных торгов. Всего загружается 15 000 записей, включая данные об объемах сделок. Стандартный биржевой тикер Apple — AAPL, но в классификации RIC (Reuters Instrument Code) он обозначается как AAPL.O.

```
In [26]: import eikon as ek ❶
```

```
In [27]: data = ek.get_timeseries('AAPL.O', fields='*',  
                                start_date='2018-10-18 16:00:00',  
                                end_date='2018-10-18 17:00:00',  
                                interval='tick') ❷
```

```
In [28]: data.info() ❸  
<class 'pandas.core.frame.DataFrame'  
DatetimeIndex: 35350 entries, 2018-10-18 16:00:00.002000 to  
                2018-10-18 16:59:59.888000  
Data columns (total 2 columns):  
VALUE      35285 non-null float64  
VOLUME     35350 non-null float64  
dtypes: float64(2)  
memory usage: 828.5 KB
```

```
In [29]: data.tail() ❹
```

```
Out[29]:
```

	AAPL.O	VALUE	VOLUME
	Date		
	2018-10-18 16:59:59.433	217.13	10.0
	2018-10-18 16:59:59.433	217.13	12.0
	2018-10-18 16:59:59.439	217.13	231.0
	2018-10-18 16:59:59.754	217.14	100.0
	2018-10-18 16:59:59.888	217.13	100.0

- ❶ Для работы с программным интерфейсом Eikon требуется подписка на службу и наличие подключения к серверу.
- ❷ Получение котировок акций Apple (AAPL.O).
- ❸ Вывод последних пяти записей из таблицы котировок.

Программный интерфейс Eikon позволяет получить доступ не только к структурированным финансовым данным, например к исторической информации о стоимости акций, но и к неструктурированным данным, таким как *новости*. В следующем примере показано, как получить метаданные

небольшой подборки новостей и вывести на экран начальный фрагмент одной из статей.

```
In [30]: news = ek.get_news_headlines('R:AAPL.O Language:LEN',  
                                       date_from='2018-05-01',  
                                       date_to='2018-06-29',  
                                       count=7) ❶
```

```
In [31]: news ❶
```

```
Out[31]:
```

			versionCreated \			text \			storyId \				sourceCode
2018-06-28	23:00:00.000	2018-06-28	23:00:00.000										
2018-06-28	21:23:26.526	2018-06-28	21:23:26.526										
2018-06-28	19:48:32.627	2018-06-28	19:48:32.627										
2018-06-28	17:33:10.306	2018-06-28	17:33:10.306										
2018-06-28	17:33:07.033	2018-06-28	17:33:07.033										
2018-06-28	17:31:44.960	2018-06-28	17:31:44.960										
2018-06-28	17:00:00.000	2018-06-28	17:00:00.000										
2018-06-28	23:00:00.000					RPT-FOCUS-AI ambulances and robot doc...			urn:newsml:reuters.com:20180628:nL4N1TU4F8:6				NS:RTRS
2018-06-28	21:23:26.526					Why Investors Should Love Apple's (AA...			urn:newsml:reuters.com:20180628:nNRA6e2vft:1				NS:ZACKSC
2018-06-28	19:48:32.627					Reuters Insider - Trump: We're reclai...			urn:newsml:reuters.com:20180628:nRTV1vNw1p:1				NS:CNBC
2018-06-28	17:33:10.306					Apple v. Samsung ends not with a whim...			urn:newsml:reuters.com:20180628:nNRA6e1oza:1				NS:WALLST
2018-06-28	17:33:07.033					Apple's trade-war discount extended f...			urn:newsml:reuters.com:20180628:nNRA6e1pmv:1				NS:WALLST
2018-06-28	17:31:44.960					Other Products: Apple's fast-growing ...			urn:newsml:reuters.com:20180628:nNRA6e1m3n:1				
2018-06-28	17:00:00.000					Pokemon Go creator plans to sell the ...			urn:newsml:reuters.com:20180628:nL1N1TU0PC:3				

```
In [32]: story_html = ek.get_news_story(news.iloc[1, 2]) ❷
```

```
In [33]: from bs4 import BeautifulSoup ❸
```

```
In [34]: story = BeautifulSoup(story_html, 'html5lib').get_text() ❹
```

```
In [35]: print(story[83:958]) ❺
```

```
Jun 28, 2018 For years, investors and Apple AAPL have been beholden to the iPhone, which is hardly a negative since its flagship product is largely responsible for turning Apple into one of the world's biggest companies. But Apple has slowly pushed into new growth areas, with streaming television its newest frontier. So let's take a look at what Apple has planned as it readies itself to compete against the likes of Netflix NFLX and Amazon AMZN in the battle for the new age of entertainment. Apple's second-quarter revenues jumped by 16% to reach $61.14 billion, with iPhone revenues up 14%. However, iPhone unit sales climbed only 3% and iPhone revenues accounted for over 62% of total Q2 sales. Apple knows this is not a sustainable business model, because rare is the consumer product that can remain in vogue for decades. This is why Apple has made a big push into news,
```

- ❶ Получение метаданных небольшой подборки новостей.
- ❷ Получение полного текста отдельной статьи в виде HTML-документа.
- ❸ Импорт библиотеки BeautifulSoup, отвечающей за синтаксический анализ HTML-файлов.
- ❹ Извлечение содержимого статьи в текстовом виде.
- ❺ Вывод начала статьи.

Несмотря на простоту, эти примеры демонстрируют, насколько легко с помощью пакетов Python и служб подписки можно получить доступ к структурированным и неструктурированным историческим финансовым данным. Иногда такого рода информация бывает доступна даже бесплатно через торговые площадки наподобие FXCM (будет рассматриваться в главах 14 и 16). Откуда бы ни поступали наборы данных, после успешного импорта к ним можно применять весь спектр аналитических инструментов Python.



## Финансовые системы, управляемые данными

Современные финансовые системы управляются данными. Даже крупнейшие и самые прибыльные хедж-фонды делают акцент на данных, а не собственно на финансах. Финансовые данные становятся доступными во все более крупных масштабах, причем как для коммерческих клиентов, так и для рядовых пользователей. В свою очередь, Python является языком выбора для взаимодействия с соответствующими программными интерфейсами и анализа получаемых данных.

## Финансовые системы на основе искусственного интеллекта

Благодаря доступности больших объемов финансовых данных через программные интерфейсы стало возможным применять методы *искусственного интеллекта*, а также *машинного и глубокого обучения* для решения финансовых задач, таких как, например, алгоритмическая торговля.

Python — идеальный язык для задач искусственного интеллекта, к которому в первую очередь обращаются исследователи и разработчики. В этом смысле финансовая отрасль получает дополнительные преимущества, пользуясь имеющимися наработками в самых разных областях, иногда совершенно не связанных с финансами. В качестве примера можно взять открытую нейросетевую библиотеку TensorFlow ([www.tensorflow.org](http://www.tensorflow.org)), разработанную и сопровождаемую компанией Google. Эта же библиотека применяется компанией Alphabet (владельцем Google) для разработки беспилотных автомобилей.

Несмотря на то что цели TensorFlow даже косвенно не связаны с задачами автоматизированной алгоритмической биржевой торговли, с помощью TensorFlow можно, например, предсказывать волатильность финансовых рынков (соответствующие примеры будут приведены в главе 15).

Один из самых популярных пакетов для машинного обучения — Scikit-learn. В следующем примере упрощенно показано, как применять алгоритмы классификации для предсказания того, в каком направлении будут меняться рыночные цены, и как на основе этих предсказаний построить стратегию алгоритмической торговли. Все детали будут объясняться в главе 15, поэтому пример дан в максимально компактном виде. Для начала необходимо импортировать данные и подготовить массив признаков (накопительные показатели логарифмической доходности с векторами смещений).

```
In [36]: import numpy as np
import pandas as pd
```

```
In [37]: data = pd.read_csv('.././source/tr_eikon_eod_data.csv',
                             index_col=0, parse_dates=True)
        data = pd.DataFrame(data['AAPL.O']) ❶
        data['Доходность'] = np.log(data / data.shift()) ❷
        data.dropna(inplace=True)
```

```
In [38]: lags = 6
```

```
In [39]: cols = []
        for lag in range(1, lags + 1):
            col = 'lag_{}'.format(lag)
            data[col] = np.sign(data['Доходность'].shift(lag)) ❸
            cols.append(col)
        data.dropna(inplace=True)
```

- ❶ Получение исторических котировок акций компании Apple (AAPL.O).
- ❷ Вычисление логарифмической доходности за весь период.
- ❸ Генерирование столбцов объекта DataFrame и заполнение их направленными показателями логарифмической доходности (+1 или -1).

Далее создается объект модели для *метода опорных векторов* (Support Vector Machine — SVM), после чего выполняется обучение модели и строится прогноз. На рис. 1.2 можно увидеть, что торговая стратегия, основанная на полученном прогнозе, применительно к акциям компании Apple дает лучший результат, чем инвестиция на основе простой доходности.

```
In [40]: from sklearn.svm import SVC
```

```
In [41]: model = SVC(gamma='scale') ❶
```

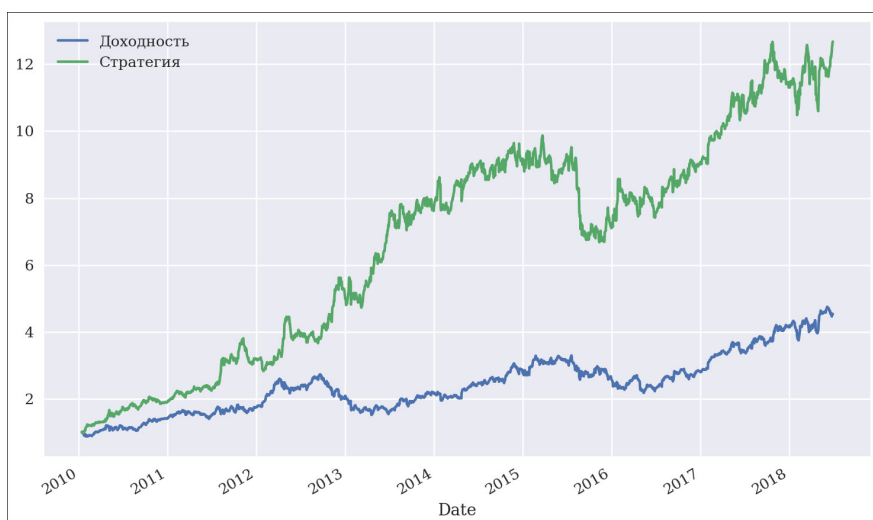
```
In [42]: model.fit(data[cols], np.sign(data['Доходность'])) ❷
Out[42]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovr', degree=3, gamma='scale',
            kernel='rbf', max_iter=-1, probability=False,
            random_state=None, shrinking=True, tol=0.001,
            verbose=False)
```

```
In [43]: data['Прогноз'] = model.predict(data[cols]) ❸
```

```
In [44]: data['Стратегия'] = data['Прогноз'] * data['Доходность'] ❹
```

```
In [45]: data[['Доходность', 'Стратегия']].cumsum().apply(np.exp).plot(
        figsize=(10, 6)); ⑤
```

- ❶ Создание объекта модели.
- ❷ Обучение модели на основе имеющихся признаков и размеченных данных (с учетом векторов направлений).
- ❸ Применение обученной модели для создания прогнозов (в пределах выборки), которые становятся позициями торговой стратегии.
- ❹ Вычисление логарифмической доходности по торговой стратегии с учетом прогнозных значений и эталонных значений доходности.
- ❺ Построение графика доходности торговой стратегии, основанной на алгоритмах машинного обучения, в сравнении с инвестицией на основе простой доходности.



*Рис. 1.2. Доходность торговой стратегии, полученной с помощью алгоритмов машинного обучения, в сравнении с инвестицией в акции компании Apple на основе простой доходности*

Конечно, в этом упрощенном примере не учитываются транзакционные издержки, а исходный набор данных не разделяется на обучающий и тестовый наборы. Тем не менее он наглядно показывает, насколько легко применять алгоритмы машинного обучения к финансовым данным, по крайней мере, с технической точки зрения (на практике придется решить еще ряд важных задач).



## Финансовые системы на основе искусственного интеллекта

Технологии искусственного интеллекта вызывают такие же изменения в финансовой отрасли, как и в других областях. Доступность огромных объемов финансовых данных через программные интерфейсы служит катализатором изменений. Базовые методы искусственного интеллекта, а также машинного и глубокого обучения будут рассмотрены в главе 13, а в главах 15 и 16 мы научимся применять их в алгоритмической торговле. Но учитывайте, что применение искусственного интеллекта в финансовых расчетах — тема отдельной книги.

Финансовые системы на основе искусственного интеллекта как естественное продолжение финансовых систем, управляемых данными, — привлекательное направление как для исследователей, так и для разработчиков. В книге описывается применение целого ряда методик искусственного интеллекта и машинного обучения в различных контекстах, но все же основная тема книги, как следует из ее названия, — фундаментальные методы работы с финансовыми данными с помощью базовых инструментов Python. Эти методы имеют не меньшее значение и в системах искусственного интеллекта.

## Резюме

Язык программирования Python — в особенности его экосистема — служит идеальной технологической базой для разработки финансовых приложений, причем это касается как индустрии в целом, так и отдельных проектов. Язык обладает целым рядом преимуществ, среди которых понятный синтаксис, эффективные подходы к разработке приложений, а также пригодность для разработки как прототипов, так и окончательных версий приложений в рамках единого производственного цикла. Наличие большого количества свободно доступных пакетов, библиотек и функциональных средств позволяет решать большинство задач, возникающих в современной финансовой среде. Язык соответствует всем требованиям, выдвигаемым с точки зрения анализа данных, работы с большими данными, производительности вычислений, соответствия регуляторным ограничениям и технологическим стандартам. Экосистема Python образует целостную, полнофункциональную среду разработки полного цикла, соответствующую целям даже крупных финансовых организаций.

Кроме того, Python стал языком выбора в системах искусственного интеллекта, как в целом, так и в частных проектах машинного и глубокого обучения.

Таким образом, это правильный язык для создания финансовых приложений, управляемых данными, а также финансовых систем на основе искусственного интеллекта — два ключевых направления, которые будут определять перспективы финансовой отрасли в обозримом будущем.

## Дополнительные ресурсы

В следующих книгах более глубоко освещаются темы, поверхностно затронутые в данной главе (в частности, инструменты Python, анализ деривативов, машинное обучение и его применение в финансовом анализе).

- Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).
- Lopez de Prado, Marcos. *Advances in Financial Machine Learning* (2018, Wiley).
- VanderPlas, Jake. *Python Data Science Handbook* (2016, O'Reilly).

Что касается алгоритмической торговли, то на сайте автора книги предлагается ряд обучающих курсов, посвященных применению Python и ряда других программных инструментов в этой быстро развивающейся сфере:

- <http://pyalgo.tpq.io>
- <http://certificate.tpq.io>

В главе цитировались следующие источники.

1. Ding, Cubillas. *Optimizing the OTC Pricing and Valuation Infrastructure* (2010, Celent).
2. Lewis, Michael. *Flash Boys* (2014, W. W. Norton & Company).
3. Patterson, Scott. *The Quants* (2010, Crown Business).





---

# Инфраструктура Python

Мастерство плотника проявляется в том, что его работа сделана аккуратно и детали хорошо подогнаны. При этом все, что он сделал, а не только отдельные части его работы, должно соответствовать плану. Это очень важно.

*Миямото Мусаси “Книга пяти колец”*

Для новичков развертывание среды Python выглядит запутанным процессом. То же самое касается установки многочисленных библиотек и пакетов. Прежде всего, существует множество реализаций Python, включая CPython, Jython, IronPython и PyPy. Кроме того, необходимо учитывать водораздел между платформами 2.7 и 3.x<sup>1</sup>.

Но даже если вы определились с версией Python, развертывание инфраструктуры усложняется целым рядом дополнительных обстоятельств.

- Интерпретатор языка (в нашем случае — установочный пакет CPython) исходно поставляется только с так называемой *стандартной библиотекой* (она, в частности, включает наиболее распространенные математические функции)
- Дополнительные пакеты (а их огромное множество) нужно устанавливать отдельно.
- Самостоятельная компиляция/сборка нестандартных пакетов — непростая задача, поскольку необходимо учитывать межпакетные зависимости и требования конкретной операционной системы.
- Последующий учет межпакетных зависимостей и поддержка согласованности версий (т.е. обслуживание инфраструктуры) — зачастую утомительный и трудоемкий процесс.
- Обновление некоторых пакетов приводит к тому, что приходится перекомпилировать множество других пакетов.

---

<sup>1</sup> В книге рассматривается версия CPython 3.7, которая была текущей на момент подготовки издания. Это наиболее популярная версия языка.

- Обновление или замена одного из пакетов может нарушить работу других программ.

К счастью, в нашем распоряжении имеются специальные инструменты, призванные помочь в преодолении указанных трудностей. В данной главе мы рассмотрим следующие технологии развертывания среды Python.

### *Менеджеры пакетов*

Менеджеры пакетов наподобие `pip` (<https://pypi.org/project/pip/>) и `conda` (<https://conda.io/en/latest/intro.html>) предназначены для установки, обновления и удаления пакетов Python. Они также помогают контролировать согласованность версий различных пакетов.

### *Менеджеры виртуального окружения*

Менеджеры виртуального окружения, такие как `virtualenv` (<https://pypi.org/project/virtualenv/>) и `conda`, позволяют одновременно управлять несколькими дистрибутивами Python (например, 2.7 и 3.7, установленными в одной системе) и безопасно тестировать новые релизы для разработчиков, не опасаясь нарушить рабочую среду<sup>2</sup>.

### *Контейнеры*

Контейнеры Docker — это автономные файловые системы, содержащие все элементы, которые требуются для запуска определенной программы, включая программный код, динамические библиотеки и системные утилиты. Например, можно запустить операционную систему Ubuntu 18.04 с установленным в ней дистрибутивом Python 3.7 и соответствующим программным кодом Python в контейнере Docker, который запущен на компьютере с macOS или Windows 10.

### *Облачные экземпляры*

Для развертывания финансовых приложений, написанных на Python, нужна надежная, защищенная и высокопроизводительная среда. Таким требованиям зачастую отвечает только профессиональная облачная инфраструктура, предлагаемая в виде облачных экземпляров различного масштаба. Одним из преимуществ облачного экземпляра (виртуального сервера) по сравнению с выделенным сервером является то, что пользователи обычно платят только за операционное время. Другое преимущество заключается в том, что в случае необходимости облачный экземпляр можно арендовать чуть ли не на минуту, что позволяет создавать гибкие масштабируемые решения.

---

<sup>2</sup> Утилита `pipenv` (<https://github.com/pyupio/pipenv>) совмещает возможности менеджера пакетов `pip` и менеджера виртуального окружения `virtualenv`.

Цель главы — помочь вам установить и настроить рабочую среду Python, оснастив ее всеми необходимыми профессиональными инструментами анализа данных и пакетами визуализации. Эта среда послужит фундаментом для запуска кода Python в последующих главах.

## conda как менеджер пакетов

Несмотря на то что утилиту conda можно установить автономно, лучше воспользоваться компактным дистрибутивом Miniconda, который включает conda как менеджер пакетов и виртуального окружения.

### Установка Miniconda

Дистрибутив Miniconda доступен для Windows, macOS и Linux. Нужную версию можно загрузить с официального сайта (<https://docs.conda.io/en/latest/miniconda.html>). В следующем примере мы запускаем сеанс в Docker-контейнере на основе Ubuntu, загружая 64-разрядный инсталлятор Linux через утилиту wget и устанавливая Miniconda. Данный пример (возможно, с незначительными изменениями) должен работать в любой системе на основе Linux или macOS.

```
$ docker run -ti -h py4fi -p 11111:11111 ubuntu:latest /bin/bash

root@py4fi:/# apt-get update; apt-get upgrade -y
...
root@py4fi:/# apt-get install -y bzip2 gcc wget
...
root@py4fi:/# cd root
root@py4fi:~# wget \
> https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh \
> -O miniconda.sh
...
HTTP request sent, awaiting response... 200 OK
Length: 62574861 (60M) [application/x-sh]
Saving to: 'miniconda.sh'

miniconda.sh          100%[=====] 59.68M  5.97MB/s   in 11s

2018-09-15 09:44:28 (5.42 MB/s) - 'miniconda.sh' saved
[62574861/62574861]

root@py4fi:~# bash miniconda.sh
```

Welcome to Miniconda3 4.5.11

In order to continue the installation process, please review the license agreement.

Please, press ENTER to continue

>>>

Чтобы начать процесс установки, нажмите клавишу <Enter>. Подтвердите условия лицензионного соглашения, введя **yes**.

...

Do you accept the license terms? [yes|no]

[no] >>> **yes**

Miniconda3 will now be installed into this location:

/root/miniconda3

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

[/root/miniconda3] >>>

PREFIX=/root/miniconda3

installing: python-3.7. ...

...

installing: requests-2.19.1-py37\_0 ...

installing: conda-4.5.11-py37\_0 ...

installation finished.

Далее необходимо подтвердить путь установки и разрешить Miniconda добавлять его в переменную среды PATH. Еще раз введите **yes**.

Do you wish the installer to prepend the Miniconda3 install location to PATH in your /root/.bashrc ? [yes|no]

[no] >>> **yes**

Appending source /root/miniconda3/bin/activate to /root/.bashrc

A backup will be made to: /root/.bashrc-miniconda3.bak

For this change to become active, you have to open a new terminal.

Thank you for installing Miniconda3!

root@py4fi:~#

Теперь можно обновить утилиту conda и среду Python<sup>3</sup>.

```
root@py4fi:~# export PATH="/root/miniconda3/bin/:$PATH"
root@py4fi:~# conda update -y conda python
...
root@py4fi:~# echo ". /root/miniconda3/etc/profile.d/conda.sh" >>
~/.bashrc
root@py4fi:~# bash
```

Выполнив указанные действия, вы получите в свое распоряжение базовый интерпретатор Python и менеджер пакетов conda. В базовый дистрибутив Python включен ряд полезных библиотек, например СУБД SQLite3 (<https://sqlite.org>). Попробуйте запустить Python, чтобы проверить, находит ли система путь к интерпретатору.

```
root@py4fi:~# python
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
[GCC 7.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello Python for Finance World.')
Hello Python for Finance World.
>>> exit()
root@py4fi:~#
```

## Выполнение основных команд в менеджере conda

Утилита conda применяется для установки, обновления и удаления любых пакетов Python. Ниже приведен синтаксис основных команд.

*Установка Python x.x*

```
conda install python=x.x
```

*Обновление Python*

```
conda update python
```

*Установка пакета*

```
conda install $ИМЯ_ПАКЕТА
```

*Обновление пакета*

```
conda update $ИМЯ_ПАКЕТА
```

*Удаление пакета*

```
conda remove $ИМЯ_ПАКЕТА
```

---

<sup>3</sup> Пакет Miniconda обновляется намного реже, чем conda и Python.

Обновление самой утилиты *conda*

```
conda update conda
```

Поиск пакета

```
conda search $КРИТЕРИЙ_ПОИСКА
```

Вывод списка установленных пакетов

```
conda list
```

К примеру, для установки NumPy — одной из важнейших библиотек научного стека Python — достаточно ввести одну-единственную команду. Если библиотека устанавливается на компьютере, который оснащен процессором Intel, то вместе с NumPy автоматически устанавливается математическая библиотека *mkl* (<https://docs.continuum.io/mkl-optimizations/>), ускоряющая вычисления не только для NumPy, но и для многих других пакетов научного стека<sup>4</sup>.

```
root@py4fi:~# conda install numpy
```

```
Solving environment: done
```

```
## Package Plan ##
```

```
environment location: /root/miniconda3
```

```
added / updated specs:
```

```
- numpy
```

The following packages will be downloaded:

package	build	
----- -----		
mkl-2019.0	117	204.4 MB
intel-openmp-2019.0	117	721 KB
mkl_random-1.0.1	py37h4414c95_1	372 KB
libgfortran-ng-7.3.0	hdf63c60_0	1.3 MB
numpy-1.15.1	py37h1d66e8a_0	37 KB
numpy-base-1.15.1	py37h81de0dd_0	4.2 MB
blas-1.0	mkl	6 KB
mkl_fft-1.0.4	py37h4414c95_1	149 KB
----- -----		
	Total:	211.1 MB

---

<sup>4</sup> При установке метапакета *nomkl* с помощью команды `conda install numpy nomkl` библиотека *mkl* и связанные с ней пакеты не будут устанавливаться автоматически.

The following NEW packages will be INSTALLED:

```
blas:          1.0-mkl
intel-openmp:  2019.0-117
libgfortran-ng: 7.3.0-hdf63c60_0
mkl:           2019.0-117
mkl_fft:       1.0.4-py37h4414c95_1
mkl_random:    1.0.1-py37h4414c95_1
numpy:         1.15.1-py37h1d66e8a_0
numpy-base:    1.15.1-py37h81de0dd_0
```

Proceed ([y]/n)? y

Downloading and Extracting Packages

```
mkl-2019.0      | 204.4 MB | ##### | 100%
...
numpy-1.15.1    | 37 KB   | ##### | 100%
numpy-base-1.15.1 | 4.2 MB  | ##### | 100%
...
root@py4fi:~#
```

За один раз можно установить сразу несколько пакетов. Флаг -y означает, что на все возникающие вопросы следует отвечать yes.

```
root@py4fi:~# conda install -y ipython matplotlib pandas pytables \
> scikit-learn scipy
```

```
...
pytables-3.4.4  | 1.5 MB  | ##### | 100%
kiwisolver-1.0.1 | 83 KB   | ##### | 100%
icu-58.2        | 22.5 MB | ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
root@py4fi:~#
```

После завершения инсталляции в системе станут доступны наиболее важные библиотеки, применяемые для финансового анализа. Рассмотрим их назначение.

### *IPython*

Улучшенная интерактивная оболочка Python.

### *matplotlib*

Стандартная библиотека визуализации данных в Python.



## *NumPy*

Библиотека для работы с числовыми массивами.

## *pandas*

Библиотека, предназначенная для управления табличными данными, например временными рядами.

## *PyTables*

Оболочка для работы с файлами формата HDF5 (<http://hdfgroup.org/>).

## *Scikit-learn*

Пакет инструментов машинного обучения.

## *SciPy*

Библиотека классов и функций, применяемых в научных расчетах (устанавливается в связке с другими библиотеками).

В результате вы получите базовый набор инструментов анализа данных, в том числе финансовых. В следующем примере показано, как с помощью IPython получить список псевдослучайных чисел, сгенерированных библиотекой NumPy.

```
root@py4fi:~# ipython
```

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
```

```
Type 'copyright', 'credits' or 'license' for more information
```

```
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]: import numpy as np
```

```
In [2]: np.random.seed(100)
```

```
In [3]: np.random.standard_normal((5, 4))
```

```
Out[3]:
```

```
array([[ -1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],  
       [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],  
       [-0.18949583,  0.25500144, -0.45802699,  0.43516349],  
       [-0.58359505,  0.81684707,  0.67272081, -0.10441114],  
       [-0.53128038,  1.02973269, -0.43813562, -1.11831825]])
```

```
In [4]: exit
```

```
root@py4fi:~#
```

С помощью команды `conda list` можно получить список установленных пакетов.

```
root@py4fi:~# conda list
# packages in environment at /root/miniconda3:
#
# Name                      Version                Build    Channel
asn1crypto                  0.24.0                 py37_0
backcall                    0.1.0                 py37_0
blas                        1.0                    mkl
blosc                       1.14.4                hdbcaa40_0
bzip2                       1.0.6                 h14c3975_5
...
python                      3.7.0                 hc3d631a_0
...
wheel                      0.31.1                py37_0
xz                          5.2.4                 h14c3975_4
yaml                       0.1.7                 had09818_2
zlib                        1.2.11                ha838bed_2
root@py4fi:~#
```

Если пакет больше не нужен, удалите его с помощью команды `conda remove`.

```
root@py4fi:~# conda remove scikit-learn
Solving environment: done
```

```
## Package Plan ##
```

```
environment location: /root/miniconda3
removed specs:
  - scikit-learn
```

The following packages will be REMOVED:

```
scikit-learn: 0.19.1-py37hedc7406_0
```

```
Proceed ([y]/n)? y
```

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
root@py4fi:~#
```

Преимущества утилиты `conda` как менеджера пакетов вполне очевидны. Но в полной мере ее возможности раскрываются при использовании в качестве менеджера виртуального окружения.



### Простое управление пакетами

Использование утилиты `conda` в качестве менеджера пакетов упрощает задачи установки, обновления и удаления пакетов Python. Вам не придется выполнять самостоятельную сборку и компиляцию пакетов, что сопряжено с целым рядом трудностей, связанных с наличием большого количества зависимостей и нюансов различных операционных систем.

## `conda` как менеджер виртуального окружения

В зависимости от выбранной версии инсталлятора Miniconda устанавливает либо Python 2.7, либо Python 3.7. При этом менеджер `conda` позволяет установить в системе сразу обе среды: например, к стандартной версии Python 3.7 добавить отдельную инсталляцию Python 2.7.x. Для этого в утилите `conda` имеются следующие команды.

*Создание виртуальной среды*

```
conda create --name $ИМЯ_СРЕДЫ
```

*Подключение виртуальной среды*

```
conda activate $ИМЯ_СРЕДЫ
```

*Отключение виртуальной среды*

```
conda deactivate $ИМЯ_СРЕДЫ
```

*Удаление виртуальной среды*

```
conda env remove --name $ИМЯ_СРЕДЫ
```

*Экспорт виртуальной среды в файл*

```
conda env export > $ИМЯ_ФАЙЛА
```

*Создание виртуальной среды из файла*

```
conda env create -f $ИМЯ_ФАЙЛА
```

*Вывод списка всех имеющихся виртуальных сред*

```
conda info --envs
```

В качестве примера создадим виртуальную среду с именем `py27`, установим в ней IPython и выполним строку кода Python 2.7.

```
root@py4fi:~# conda create --name py27 python=2.7
Solving environment: done
```

```
## Package Plan ##
```

```
environment location: /root/miniconda3/envs/py27
```

```
added / updated specs:
- python=2.7
```

```
The following NEW packages will be INSTALLED:
```

```
ca-certificates: 2018.03.07-0
...
python:          2.7.15-h1571d57_0
...
zlib:            1.2.11-ha838bed_2
```

```
Proceed ([y]/n)? y
```

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

```
#
# To activate this environment, use:
# > conda activate py27
#
# To deactivate an active environment, use:
# > conda deactivate
#
```

```
root@py4fi:~#
```

Обратите внимание на то, что после подключения виртуальной среды приглашение командной строки будет содержать ее имя (py27).

```
root@py4fi:~# conda activate py27
(py27) root@py4fi:~# conda install ipython
Solving environment: done
...
Executing transaction: done
(py27) root@py4fi:~#
```

Теперь в IPython можно выполнять команды Python 2.7.

```
(py27) root@py4fi:~# ipython
```

```
Python 2.7.15 |Anaconda, Inc.| (default, May 1 2018, 23:32:55)  
Type "copyright", "credits" or "license" for more information.
```

```
IPython 5.8.0 -- An enhanced Interactive Python.
```

```
?          -> Introduction and overview of IPython's features.
```

```
%quickref  -> Quick reference.
```

```
help       -> Python's own help system.
```

```
object?    -> Details about 'object', use 'object??' for extra details.
```

```
In [1]: print "Hello Python for Finance World!"
```

```
Hello Python for Finance World!
```

```
In [2]: exit
```

```
(py27) root@py4fi:~#
```

Этот пример показывает, что утилита `conda` позволяет установить в одной системе несколько версий Python. Также можно устанавливать разные версии определенных пакетов. Добавление еще одной среды никоим образом не влияет на существующую среду Python и другие среды, которые были установлены ранее. Список всех имеющихся сред можно вывести с помощью команды `conda env list`.

```
(py27) root@py4fi:~# conda env list
```

```
# conda environments:
```

```
#
```

```
base                               /root/miniconda3
```

```
py27                               * /root/miniconda3/envs/py27
```

```
(py27) root@py4fi:~#
```

Иногда требуется поделиться информацией о среде с другими пользователями или запустить одну и ту же среду на нескольких компьютерах. Для этого следует экспортировать список установленных пакетов в файл с помощью команды `conda env export`. По умолчанию перенос среды возможен только на компьютер с аналогичной операционной системой, поскольку в результирующем файле YAML могут быть указаны не версии сборок, а только версии пакетов.

```
(py27) root@py4fi:~# conda env export --no-builds > py27env.yml
```

```
(py27) root@py4fi:~# cat py27env.yml
```

```
name: py27
```

```
channels:
  - defaults
dependencies:
  - backports=1.0
...
  - python=2.7.15
...
  - zlib=1.2.11
prefix: /root/miniconda3/envs/py27
```

```
(py27) root@py4fi:~#
```

Зачастую виртуальную среду, которая с технической точки зрения представляет собой всего лишь структуру подкаталогов, создают для выполнения быстрых тестов<sup>5</sup>. В таком случае ее можно легко удалить сразу же после деактивации с помощью команды `conda env remove`.

```
(py27) root@py4fi:~# conda deactivate
root@py4fi:~# conda env remove -y --name py27
```

Remove all packages in environment /root/miniconda3/envs/py27:

```
## Package Plan ##
```

```
environment location: /root/miniconda3/envs/py27
```

The following packages will be REMOVED:

```
backports:                                1.0-py27_1
...
zlib:                                       1.2.11-ha838bed_2
```

```
root@py4fi:~#
```

На этом краткий обзор возможностей менеджера виртуального окружения `conda` можно считать завершенным.

---

<sup>5</sup> В официальной документации (<https://packaging.python.org/tutorials/installing-packages/#creating-virtual-environments>) сказано следующее: “Виртуальная среда Python позволяет устанавливать пакеты в изолированном каталоге для конкретного приложения, а не для всего дистрибутива”.



## Простое управление виртуальным окружением

Утилита `conda` не только позволяет управлять пакетами; она также представляет собой менеджер виртуального окружения для Python. Утилита упрощает создание различных дистрибутивов Python, позволяя иметь разные версии языка с разным набором пакетов на одном и том же компьютере, причем они никак не будут влиять друг на друга. Кроме того, с помощью утилиты `conda` можно экспортировать виртуальную среду в файл, чтобы перенести на другой компьютер либо поделиться с другими пользователями.

## Контейнеры Docker

Контейнеры Docker за последние годы произвели настоящий фурор в IT-среде. Несмотря на относительную новизну, технология прекрасно зарекомендовала себя в качестве стандартной платформы для эффективного развертывания практически любых типов приложений.

В данной книге контейнер Docker рассматривается как обособленная файловая система, включающая операционную систему (например, Ubuntu Server 18.04), среду Python, вспомогательные системные утилиты, а также все необходимые программные библиотеки и пакеты. Такой контейнер можно запустить как на локальном компьютере с Windows 10, так и в облачном экземпляре с операционной системой Linux.

В этом разделе не приводится полное описание Docker. Мы лишь вкратце поговорим о том, что это за технология и что она позволяет делать в контексте развертывания среды Python<sup>6</sup>.

## Контейнеры и образы

Прежде чем двигаться дальше, необходимо уяснить разницу между двумя ключевыми концепциями Docker. *Образ Docker* можно сравнить с классом Python. В свою очередь, *контейнер Docker* можно уподобить экземпляру соответствующего класса<sup>7</sup>.

Техническое определение понятия “образ” дано в словаре Docker (<https://docs.docker.com/glossary/>).

Образы Docker — это основа для контейнеров. Образ представляет собой упорядоченную коллекцию сведений об изменениях в корневой файловой системе

---

<sup>6</sup> Подробное введение в технологию Docker дано у Маттиаса и Кейна [1].

<sup>7</sup> Терминология объектно-ориентированного программирования рассматривается в главе 6.

и параметрах выполнения программ, запускаемых из контейнера. Образ обычно содержит группу файловых систем с каскадно-объединенным монтированием. Базовый образ не имеет состояния и никогда не меняется.

Там же можно найти определение понятия “контейнер”, благодаря которому аналогия с классами и объектами Python становится очевидной.

Контейнер — это исполняемый экземпляр образа Docker. Он содержит сам образ, исполняющую среду и стандартный набор инструкций.

Установка программного обеспечения Docker зависит от операционной системы. Именно поэтому мы не будем детально ее рассматривать. Дополнительные сведения можно найти на сайте <https://docs.docker.com/get-docker/>.

## Создание образа Docker с Ubuntu и Python

В этом разделе описывается процедура создания образа Docker на основе последней версии операционной системы Ubuntu, в которой установлены Miniconda и ряд важных пакетов Python. При этом контейнер обновляет список пакетов Linux, обновляет сами пакеты в случае необходимости и устанавливает дополнительные системные утилиты. Для этого применяются два сценария. Первый запускается из командной оболочки *bash* и отвечает за конфигурирование среды Linux<sup>8</sup>. Второй сценарий называется *Dockerfile* — он управляет процедурой сборки самого образа.

Установочный *bash*-сценарий, приведенный в листинге 2.1, состоит из трех разделов. В первом разделе выполняется конфигурирование среды Linux, во втором — устанавливается Miniconda, в третьем — инсталлируются дополнительные пакеты Python. Остальные детали описаны в комментариях к файлу.

### Листинг 2.1. Сценарий установки Python и дополнительных пакетов (*install.sh*)

```
#!/bin/bash
#
# Сценарий установки системных
# утилит Linux и основных
# пакетов Python
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
```

---

<sup>8</sup> Описание возможностей оболочки *bash* см. у Роббинса [2]. Посетите также сайт <https://www.gnu.org/software/bash>.



```

#
# КОНФИГУРИРОВАНИЕ LINUX
apt-get update          # обновление списка пакетов
apt-get upgrade -y      # обновление самих пакетов
# Установка системных утилит
apt-get install -y bzip2 gcc git htop screen vim wget
apt-get upgrade -y bash # обновление bash
apt-get clean           # очистка списка пакетов

# УСТАНОВКА MINICONDA
# Загрузка Miniconda
wget https://repo.continuum.io/miniconda/
    Miniconda3-latest-Linux-x86_64.sh -O Miniconda.sh
bash Miniconda.sh -b     # установка
rm -rf Miniconda.sh      # удаление инсталлятора
export PATH="/root/miniconda3/bin:$PATH" # добавление нового пути

# УСТАНОВКА БИБЛИОТЕК PYTHON
conda update -y conda python # обновление утилиты Conda и Python
conda install -y pandas      # установка библиотеки Pandas
conda install -y ipython     # установка оболочки IPython

```

Сценарий *Dockerfile*, приведенный в листинге 2.2, использует *bash*-сценарий из листинга 2.1 для сборки нового образа Docker. Выполняемые действия также описаны в комментариях.

### **Листинг 2.2. Сценарий сборки образа (Dockerfile)**

```

#
# Создание образа Docker, включающего последнюю версию
# операционной системы Ubuntu и базовый дистрибутив Python
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#

# Использование последней версии Ubuntu
FROM ubuntu:latest

# Сведения об установщике
MAINTAINER yves

# Подключение bash-сценария
ADD install.sh /

```

```
# Изменение прав доступа к сценарию
RUN chmod u+x /install.sh

# Выполнение bash-сценария
RUN /install.sh

# Добавление нового пути
ENV PATH /root/miniconda3/bin:$PATH

# Запуск оболочки IPython при выполнении контейнера
CMD ["ipython"]
```

Если оба этих файла находятся в одной папке, а в системе установлено программное обеспечение Docker, то процесс сборки нового образа Docker предельно упрощается. В данном примере образ помечается дескриптором `py4fi:basic`. Его нужно указывать в качестве ссылки на образ при запуске соответствующего контейнера.

```
~/Docker$ docker build -t py4fi:basic .
```

```
Removing intermediate container 5fec0c9b2239
---> accee128d9e9
Step 6/7 : ENV PATH /root/miniconda3/bin:$PATH
---> Running in a2bb97686255
Removing intermediate container a2bb97686255
---> 73b00c215351
Step 7/7 : CMD ["ipython"]
---> Running in ec7acd90c991
Removing intermediate container ec7acd90c991
---> 6c36b9117cd2
Successfully built 6c36b9117cd2
Successfully tagged py4fi:basic
~/Docker$
```

Получить список доступных образов Docker можно с помощью команды `docker images`. Последний созданный образ указывается в списке первым.

```
(py4fi) ~/Docker$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
py4fi	basic	f789dd230d6f	About a minute ago	1.79GB
ubuntu	latest	cd6d8154f1e1	9 days ago	84.1MB

```
(py4fi) ~/Docker$
```

После успешной сборки образа `py4fi:basic` запустить соответствующий контейнер Docker можно с помощью команды `docker run`. Параметр `-ti` позволяет запустить в контейнере интерактивные процессы, например командную оболочку (см. <https://docs.docker.com/engine/reference/run/>).

```
~/Docker$ docker run -ti py4fi:basic
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]: import numpy as np
```

```
In [2]: a = np.random.standard_normal((5, 3))
```

```
In [3]: import pandas as pd
```

```
In [4]: df = pd.DataFrame(a, columns=['a', 'b', 'c'])
```

```
In [5]: df
```

```
Out[5]:
```

	a	b	c
0	-1.412661	-0.881592	1.704623
1	-1.294977	0.546676	1.027046
2	1.156361	1.979057	0.989772
3	0.546736	-0.479821	0.693907
4	-1.972943	-0.193964	0.769500

```
In [6]:
```

Выход из оболочки IPython приведет к закрытию контейнера, так как это единственное запущенное в нем приложение. Если же нужно выйти из контейнера, *не завершая его работу*, то нажмите комбинацию клавиш `<Ctrl+P>` и `<Ctrl+Q>`.

Выполнив команду `docker ps`, мы увидим, что контейнер остается запущенным (равно как и другие запущенные ранее контейнеры).

```
~/Docker$ docker ps
CONTAINER ID  IMAGE           COMMAND          CREATED          \
e815df8f0f4d  py4fi:basic    "ipython"       About a minute ago \
4518917de7dc  ubuntu:latest  "/bin/bash"     About an hour ago  \
d081b5c7add0  ubuntu:latest  "/bin/bash"     21 hours ago      \
```

```
STATUS
Up About a minute
Up About an hour
Up 21 hours
```

```
~/Docker$
```

Подключение контейнера Docker осуществляется командой `docker attach $ID`, причем в качестве идентификатора достаточно указать первые несколько символов.

```
~/Docker$ docker attach e815d
```

```
In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
a      5 non-null float64
b      5 non-null float64
c      5 non-null float64
dtypes: float64(3)
memory usage: 200.0 bytes
```

```
In [7]: exit
~/Docker$
```

Команда `exit` завершает работу IPython и вызывает закрытие контейнера Docker. Контейнер можно удалить с помощью команды `docker rm`.

```
~/Docker$ docker rm e815d
e815d
~/Docker$
```

Аналогичным образом можно удалить сам образ `py4fi:basic` с помощью команды `docker rmi`. Контейнеры достаточно компактны, но образы могут занимать много места на диске. Например, образ `py4fi:basic` имеет размер около 2 Гбайт. Вот почему необходимо регулярно очищать список образов Docker.

```
~/Docker$ docker rmi 6c36b9117cd2
```

О контейнерах Docker можно говорить еще очень долго, но читателям книги достаточно знать, что контейнеры представляют собой современную технологию развертывания полностью изолированной среды Python и с их помощью можно распространять приложения алгоритмической торговли.



## Преимущества контейнеров Docker

Если вам еще не доводилось работать с контейнерами Docker, то рекомендуем познакомиться с ними поближе. Они имеют целый ряд преимуществ, связанных с развертыванием среды Python, которые проявляются не столько в локальном окружении, сколько при работе с облачными экземплярами и серверами алгоритмической торговли.

## Облачные экземпляры

В этом разделе будет показано, как сконфигурировать полноценную инфраструктуру Python на облаке DigitalOcean ([www.digitalocean.com](http://www.digitalocean.com)). Существует множество других облачных решений, включая AWS (Amazon Web Services; <https://aws.amazon.com/ru/>). Однако DigitalOcean славится своей простотой и относительной дешевизной небольших облачных экземпляров, называемых *дроплетами*. Наименьший дроплет, которого обычно достаточно для задач интерактивной разработки, обойдется всего в 5 долл. в месяц или 0,7 цента в час. Оплата взимается на почасовой основе, что позволяет создать дроплет буквально на пару часов, выполнить в нем все необходимые действия и удалить, заплатив 1,5 цента<sup>9</sup>.

Нашей целью будет создание собственного дроплета на облаке DigitalOcean, в котором мы установим Python 3.7, ключевые пакеты (NumPy, pandas и пр.) и сервер Jupyter Notebook (<https://jupyter.org/>) с паролем доступом, снабженный SSL-шифрованием. Сервер будет содержать три базовых компонента, доступ к которым можно получить через обычный браузер.

### *Jupyter Notebook*

Популярная интерактивная среда разработки, поддерживающая несколько языков программирования (в частности, Python, R и Julia).

### *Терминал*

Командная оболочка, запускаемая из браузера и позволяющая выполнять типичные задачи системного администрирования, а также пользоваться утилитами типа Vim или git.

### *Редактор*

Запускаемый из браузера редактор кода с цветовой разметкой синтаксиса, поддержкой различных языков программирования и типов файлов.

---

<sup>9</sup> Новые пользователи, которые регистрируются по ссылке [https://bit.ly/do\\_sign\\_up](https://bit.ly/do_sign_up), получают 60-дневный приветственный бонус в размере 100 долларов.

Установка Jupyter Notebook на дроплете позволяет развернуть среду разработки в браузере, избегая необходимости регистрироваться в облачном экземпляре через SSL-соединение.

Для решения поставленных задач нам понадобится несколько файлов.

#### *Сценарий настройки сервера*

Управляет всеми этапами, в частности копирует все остальные файлы в дроплет и запускает их на выполнение.

#### *Сценарий инсталляции Python и Jupyter Notebook*

Устанавливает Python, дополнительные пакеты и среду Jupyter Notebook, а также запускает сервер Jupyter Notebook.

#### *Файл конфигурирования Jupyter Notebook*

Настраивает сервер Jupyter Notebook, в частности подключая парольный доступ.

#### *Файлы открытого и закрытого ключей RSA*

Требуются для SSL-шифрования на сервере Jupyter Notebook.

Мы пройдем этот список в обратном порядке.

## Открытый и закрытый ключи RSA

Чтобы иметь возможность создавать защищенное подключение к серверу Jupyter Notebook из любого браузера, нам нужен SSL-сертификат, состоящий из открытого и закрытого ключей RSA (<https://ru.wikipedia.org/wiki/RSA>). Как правило, такой сертификат выдается одним из центров сертификации (Certificate Authority — CA), но для задач книги будет достаточно самостоятельно сгенерированного сертификата<sup>10</sup>. Пары RSA-ключей можно сгенерировать с помощью утилиты OpenSSL ([www.openssl.org](http://www.openssl.org)). Ниже показано, как сгенерировать сертификат для доступа к серверу Jupyter Notebook (заполняйте предлагаемые поля собственными значениями).

```
~/cloud$ openssl req -x509 -nodes -days 365 -newkey \
rsa:1024 -out cert.pem -keyout cert.key
Generating a 1024 bit RSA private key
..+++++
.....+++++
writing new private key to 'cert.key'
```

---

<sup>10</sup> При работе с таким сертификатом понадобится добавить исключение в настройки безопасности браузера.

Далее вас попросят ввести информацию, включаемую в сертификат. Вам обязательно нужно будет указать *уникальное имя* (Distinguished Name — DN). Можно оставить некоторые из полей незаполненными или принять значения, заданные по умолчанию. Поле останется незаполненным, если в качестве значения ввести точку (.).

```
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Saarland
Locality Name (eg, city) []:Voelklingen
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TPQ GmbH
Organizational Unit Name (eg, section) []:Python for Finance
Common Name (e.g. server FQDN or YOUR name) []:Jupyter
Email Address []:team@tpq.io
~/cloud$ ls
cert.key      cert.pem
~/cloud$
```

Файлы *cert.key* и *cert.pem* нужно скопировать в дроплет, добавив ссылку на них в конфигурационный файл Jupyter Notebook, который рассматривается далее.

## Конфигурационный файл Jupyter Notebook

Общедоступный сервер Jupyter Notebook можно развернуть в безопасной среде согласно инструкциям, приведенным в официальной документации (<http://bit.ly/2Ka0tfI>). В частности, доступ к серверу можно защитить паролем. Для этого следует воспользоваться функцией `passwd()` из пакета `notebook.auth`, которая генерирует хеш-код пароля. В приведенном ниже примере в качестве пароля используется слово “jupyter”.

```
~/cloud$ ipython
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from notebook.auth import passwd

In [2]: passwd('jupyter')
Out[2]: 'sha1:d4d34232ac3a:55ea0ffd78cc3299e3e5e6ecc0d36be0935d424b'

In [3]: exit
```

Полученный хеш-код следует включить в конфигурационный файл сервера Jupyter Notebook (листинг 2.3). Предполагается, что файлы RSA-ключей находятся в папке */root/.jupyter/* дроплета.

### Пример 2.3. Файл конфигурации Jupyter Notebook (jupyter\_notebook\_config.py)

```
#
# Конфигурационный файл Jupyter Notebook
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
# SSL-ШИФРОВАНИЕ
# Замените пути (и сами файлы) своими вариантами
c.NotebookApp.certfile = u'/root/.jupyter/cert.pem'
c.NotebookApp.keyfile = u'/root/.jupyter/cert.key'

# IP-АДРЕС И ПОРТ
# Чтобы разрешить все IP-адреса облачного
# экземпляра, задайте '*'
c.NotebookApp.ip = '*'
# Имеет смысл использовать фиксированный
# номер порта для доступа к серверу
c.NotebookApp.port = 8888

# ЗАЩИТА ПАРОЛЕМ
# Пароль: 'jupyter'
# Замените хеш-кодом своего, более надежного пароля
c.NotebookApp.password = \
    'sha1:d4d34232ac3a:55ea0ffd78cc3299e3e5e6ecc0d36be0935d424b'

# БЛОКИРОВКА ЗАПУСКА БРАУЗЕРА
# Предотвращение запуска браузера из Jupyter
c.NotebookApp.open_browser = False
```



#### Jupyter Notebook и безопасность

Развертывание Jupyter Notebook в облаке вызывает целый ряд проблем с безопасностью, ведь это полнофункциональная среда разработки, запускаемая из браузера. Следовательно, крайне важно использовать все предусмотренные в Jupyter Notebook средства защиты, включая пароль и SSL-шифрование. Но это только начальный уровень. В зависимости от того, что конкретно делается в облачном экземпляре, могут понадобиться дополнительные меры безопасности.

Следующий шаг — установка Python и Jupyter Notebook в дроплете.



## Сценарий установки Python и Jupyter Notebook

Сценарий установки Python и Jupyter Notebook напоминает *bash*-сценарий, который рассматривался в разделе “Контейнеры Docker” и предназначался для инсталляции Python через установщик Miniconda в контейнере Docker. Однако сценарий, приведенный в листинге 2.4, требует запуска сервера Jupyter Notebook. Как и прежде, все выполняемые действия описаны в комментариях.

### Листинг 2.4. Сценарий установки Python и запуска сервера Jupyter Notebook (*install.sh*)

```
#!/bin/bash
#
# Сценарий установки системных утилит
# Linux, основных пакетов Python
# и сервера Jupyter Notebook
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
# КОНФИГУРИРОВАНИЕ LINUX
apt-get update           # обновление списка пакетов
apt-get upgrade -y      # обновление самих пакетов
# Установка системных утилит
apt-get install -y bzip2 gcc git htop screen vim wget
apt-get upgrade -y bash  # обновление bash
apt-get clean            # очистка списка пакетов

# УСТАНОВКА MINICONDA
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh \
-O Miniconda.sh
bash Miniconda.sh -b      # установка
rm Miniconda.sh          # удаление инсталлятора
# Добавление нового пути для текущего сеанса
export PATH="/root/miniconda3/bin:$PATH"
# Добавление нового пути в настройках оболочки
echo ". /root/miniconda3/etc/profile.d/conda.sh" >> ~/.bashrc
echo "conda activate" >> ~/.bashrc

# УСТАНОВКА БИБЛИОТЕК PYTHON
# В зависимости от решаемых задач могут
# понадобиться дополнительные пакеты
conda update -y conda      # обновление утилиты conda
conda create -y -n py4fi python=3.7 # создание среды
```

```

source activate py4fi          # запуск новой среды
conda install -y jupyter       # интерактивная разработка
                                # в браузере
conda install -y pytables      # оболочка для работы с файлами HDF5
conda install -y pandas        # пакет анализа данных
conda install -y matplotlib    # библиотека визуализации данных
conda install -y scikit-learn  # библиотека инструментов
                                # машинного обучения
conda install -y openpyxl      # библиотека обмена данными с Excel
conda install -y pyyaml        # библиотека обработки файлов YAML

pip install --upgrade pip      # обновление менеджера пакетов
pip install cufflinks          # визуализация данных из библиотеки pandas

# КОПИРОВАНИЕ ФАЙЛОВ И СОЗДАНИЕ КАТАЛОГОВ
mkdir /root/.jupyter
mv /root/jupyter_notebook_config.py /root/.jupyter/
mv /root/cert.* /root/.jupyter
mkdir /root/notebook
cd /root/notebook

# ЗАПУСК СЕРВЕРА JUPYTER NOTEBOOK
jupyter notebook --allow-root

# Запуск сервера в фоновом режиме:
# jupyter notebook --allow-root &

```

Этот сценарий нужно скопировать в дроплет и запустить из сценария оркестровки, описанного в следующем разделе.

## Сценарий оркестровки для процесса установки дроплета

Следующий *bash*-сценарий (листинг 2.5) самый короткий. Он отвечает за настройку дроплета, копируя в него все остальные файлы, при этом IP-адрес дроплета служит параметром сценария. В последней строке запускается сценарий *install.sh*, который в свою очередь устанавливает и запускает сервер Jupyter Notebook.

### Листинг 2.5. Сценарий оркестровки дроплета (*setup.sh*)

```

#!/bin/bash
#
# Настройка дроплета DigitalOcean,
# содержащего базовый стек пакетов Python
# и сервер Jupyter Notebook

```

```
#
# Python for Finance, 2nd ed.
# (c) Dr Yves J Hilpisch
#

# ПОЛУЧЕНИЕ IP-АДРЕСА ИЗ КОМАНДНОЙ СТРОКИ
MASTER_IP=$1

# КОПИРОВАНИЕ ФАЙЛОВ
scp install.sh root@${MASTER_IP}:
scp cert.* jupyter_notebook_config.py root@${MASTER_IP}:

# ЗАПУСК СЦЕНАРИЯ ИНСТАЛЛЯЦИИ
ssh root@${MASTER_IP} bash /root/install.sh
```

Теперь можно приступить к установке. Создайте новый дроплет на сайте DigitalOcean, указав для него следующие параметры.

#### *Операционная система*

Ubuntu 18.10 x64 (последняя версия на момент подготовки книги).

#### *Аппаратная конфигурация*

Одно ядро, 1 Гбайт ОЗУ, 25 Гбайт SSD (наименьший доступный дроплет).

#### *Регион дата-центра*

Франкфурт (поскольку автор книги проживает в Германии).

#### *SSH-ключ*

Добавьте новый SSH-ключ для беспарольной аутентификации<sup>11</sup>.

#### *Имя дроплета*

Можете использовать предложенное имя или сокращение типа py4fi.

Процесс создания дроплета запускается после щелчка на кнопке Create и длится около минуты. После этого вам будет выдан IP-адрес, например 46.101.156.199, если в качестве местонахождения дата-центра был указан Франкфурт. Конфигурирование такого дроплета выполняется с помощью одной команды:

```
(py3) ~/cloud$ bash setup.sh 46.101.156.199
```

Настройка длится несколько минут и завершается получением сообщения от сервера Jupyter Notebook.

---

<sup>11</sup> Подробно процедура описана на сайте DigitalOcean (<https://do.co/2DIqnH9> и <https://do.co/2A0EAL0>).

The Jupyter Notebook is running at:  
`https://[все ip-адреса системы]:8888/`

Для получения доступа к серверу достаточно ввести в строке браузера следующий адрес (обратите внимание на префикс `https://`):

`https://46.101.156.199:8888`

Скорее всего, браузер спросит, хотите ли вы добавить исключение в настройки безопасности, после чего появится форма ввода пароля (в данном случае пароль — “jupyter”). Теперь вы готовы начать разрабатывать приложения Python в браузерной среде Jupyter Notebook. Вам также становится доступной среда IPython через окно терминала и текстовый редактор. Кроме того, из браузера можно управлять рабочими файлами: загружать, выгружать на сервер, удалять и т.п.



### Преимущества облачных решений

Комбинация технологий DigitalOcean и Jupyter Notebook предоставляет в распоряжение разработчиков Python и финансовых аналитиков профессиональную вычислительную инфраструктуру. Поставщики облачных услуг и дата-центры гарантируют физическую безопасность и непрерывную доступность виртуальных компьютеров. Это также сокращает стоимость этапа разработки, поскольку плата обычно взимается на почасовой основе, и не нужно заключать никаких долгосрочных соглашений.

## Резюме

Python — ключевой язык программирования и технологическая платформа не только для данной книги, но и для большинства ведущих финансовых компаний. В то же время развертывание среды Python может оказаться непростой и весьма трудоемкой задачей. К счастью, в последние годы появился ряд эффективных технологий, помогающих справиться с возникающими проблемами. В частности, открытый пакет `conda` служит как менеджер пакетов Python и одновременно как менеджер виртуального окружения. Контейнеры Docker представляют собой следующий шаг, позволяя легко создать полноценную файловую систему и среду выполнения в изолированной “песочнице”. Следующий шаг вперед делают облачные провайдеры типа DigitalOcean, предлагая быструю аренду вычислительных мощностей в профессионально управляемых и надежно защищенных дата-центрах с почасовой оплатой. Наличие облачного экземпляра с развернутым на нем дистрибутивом Python 3.7 и защищенным сервером Jupyter Notebook предоставляет в ваше распоряжение

профессиональную среду разработки приложений Python, в которой можно создавать проекты для работы с финансовыми данными.

## Дополнительные ресурсы

Ответы на вопросы, связанные с управлением пакетами Python, можно получить по следующим адресам:

- страница менеджера пакетов `pip` (<https://pypi.org/project/pip/>);
- страница менеджера пакетов `conda` (<https://conda.io/en/latest/>);
- инструкции по установке пакетов Python (<https://packaging.python.org/tutorials/installing-packages/>).

Информацию об управлении виртуальным окружением можно найти на следующих сайтах:

- сайт менеджера виртуального окружения `virtualenv` (<https://pypi.org/project/virtualenv/>);
- сайт менеджера виртуального окружения `conda` (<https://conda.io/projects/conda/en/latest/user-guide/concepts/environments.html>);
- репозиторий менеджера пакетов и виртуального окружения `pipenv` (<https://github.com/pya/pipenv>).

Подробные сведения о технологии Docker доступны на официальном сайте ([www.docker.com](http://www.docker.com)).

О создании защищенного общедоступного сервера Jupyter Notebook рассказано в документации:

[https://jupyter-notebook.readthedocs.io/en/latest/public\\_server.html](https://jupyter-notebook.readthedocs.io/en/latest/public_server.html)

Одновременно управлять несколькими пользователями сервера Jupyter Notebook можно через специальный хаб:

<https://jupyterhub.readthedocs.io/en/stable/>

Чтобы подписаться на службу DigitalOcean, посетите страницу [http://bit.ly/do\\_sign\\_up](http://bit.ly/do_sign_up). Вы получите приветственный бонус в размере 100 долларов, которого хватит на два месяца управления самым маленьким из дроплетов.

В главе упоминались следующие источники.

1. Matthias, Karl, and Sean Kane. *Docker: Up and Running* (2015, O'Reilly).
2. Роббинс, Арнольд. *Bash. Карманный справочник, 2-е издание* (2017, Вильямс).

---

# Основы Python

Эта часть книги посвящена основам программирования на Python. Приведенная здесь информация фундаментальна для понимания всех последующих глав книги.

Главы данной части посвящены конкретным темам и послужат справочным руководством, к которому читатели смогут обращаться в случае необходимости.

- **Глава 3** посвящена типам данных и структурам Python.
- **Глава 4** посвящена библиотеке NumPy и ее классу `ndarray`.
- **Глава 5** посвящена библиотеке pandas и ее классу `DataFrame`.
- **Глава 6** посвящена объектно-ориентрованному программированию на Python.



# Типы данных и структуры Python

Плохие программисты беспокоятся о коде. Хорошие программисты беспокоятся о структурах данных и их взаимосвязях.

*Линус Торвальдс*

В главе рассматриваются следующие темы.

## Основные типы данных

В этом разделе описаны такие базовые типы данных, как `int`, `float`, `bool` и `str`.

## Основные структуры данных

Из этого раздела вы узнаете о фундаментальных структурах Python (таких, как объекты `list`). Кроме того, здесь рассматриваются управляющие конструкции, принципы функционального программирования и анонимные функции.

Цель главы — дать общую информацию о существующих в Python типах данных и структурах. Читатели, знакомые с другими языками программирования, такими как C или Matlab, легко поймут, какие особенности характерны для Python. Приемы программирования, которые будут описаны в этой главе, важны для понимания последующих глав.

В главе рассматриваются следующие типы данных и структуры.

Тип данных	Описание	Пример
<code>int</code>	Целочисленное значение	Натуральные числа
<code>float</code>	Число с плавающей точкой	Вещественные числа
<code>bool</code>	Булево значение	Истина или ложь
<code>str</code>	Строковый объект	Символ, слово, текст
<code>tuple</code>	Неизменяемая последовательность	Фиксированный набор объектов, кортеж
<code>list</code>	Изменяемая последовательность	Настраиваемый список объектов
<code>dict</code>	Изменяемая последовательность	Словарь, индексируемый с помощью ключей
<code>set</code>	Изменяемая последовательность	Набор уникальных объектов



# Основные типы данных

Python — язык с *динамической типизацией*. Это означает, что интерпретатор определяет тип объекта на этапе выполнения программы. Компилируемые языки наподобие С обычно имеют *статическую типизацию*, при которой тип объекта задается заранее<sup>1</sup>.

## Целые числа

Простейший тип данных — целое число, или `int`.

```
In [1]: a = 10
        type(a)
Out[1]: int
```

Встроенная функция `type()` сообщает тип переданного ей объекта. Это может быть как встроенный тип, так и созданный в программе класс. Во втором случае возвращаемая информация зависит от описания, заложенного программистом в классе. Как известно, в Python “все что угодно является объектом”. А это значит, что даже такие простые типы данных, как `int`, обладают собственными методами. Например, метод `bit_length()` позволяет узнать количество битов, требуемых для хранения объекта `int` в памяти.

```
In [2]: a.bit_length()
Out[2]: 4
```

Чем больше значение, хранящееся в объекте `int`, тем больше места он занимает в памяти.

```
In [3]: a = 1000000
        a.bit_length()
Out[3]: 17
```

У различных классов и объектов столько всяких методов, что запомнить их совершенно невозможно. Во многие интерпретаторы, в частности в IPython, встроены интерактивные подсказки, позволяющие узнать все методы, поддерживаемые объектом. Чтобы получить подсказку, достаточно ввести имя объекта, точку и нажать клавишу `<Tab>`. Также в Python имеется встроенная функция `dir()`, возвращающая список всех атрибутов и методов переданного ей объекта.

---

<sup>1</sup> В Cython ([www.cython.org](http://www.cython.org)) добавлена возможность статической типизации и компиляции, характерных для С. Фактически Cython — это полнофункциональный гибридный язык программирования, объединяющий возможности Python и С.

[illegible]

Python позволяет работать с очень длинными целыми числами, поскольку интерпретатор выделяет для их хранения в памяти ровно столько битов/байтов, сколько требуется.

```
In [6]: 1 + 4
Out[6]: 5
```

```
In [7]: 1 / 4
Out[7]: 0.25
```

```
In [8]: type(1 / 4)
Out[8]: float
```

Результатом выражения `1 / 4` является число `0.25`<sup>2</sup>, имеющее тип `float`. Когда к целому числу добавляется точка, например `1.` или `1.0`, интерпретатор Python начинает трактовать его как объект типа `float`. Результатом выражения, в котором один из операндов имеет тип `float`, тоже будет число с плавающей точкой<sup>3</sup>.

```
In [9]: 1.6 / 4
Out[9]: 0.4
```

<sup>2</sup> В Python 2.x по умолчанию выполняется целочисленное деление, поэтому результат будет другим. В Python 3.x целочисленное деление выполняется с помощью специального оператора (например, результатом выражения `3 // 4` будет 0).

<sup>3</sup> Термины “значение типа float” и “объект типа float” означают одно и то же, поскольку в Python все числа — *объекты*. То же самое справедливо и для любых других типов данных.

```
In [10]: type(1.6 / 4)
Out[10]: float
```

У типа `float` есть свои особенности, связанные с тем, что компьютерное представление рациональных и вещественных чисел неточное и зависит от реализации. В качестве иллюстрации определим объект `b` типа `float`. При работе с такими объектами следует учитывать, что их внутреннее представление всегда имеет определенную точность. Это становится очевидно при добавлении `0.1`, как показано ниже.

```
In [11]: b = 0.35
          type(b)
Out[11]: float

In [12]: b + 0.1
Out[12]: 0.44999999999999996
```

Причина получения столь “странного” результата заключается в том, что объекты типа `float` внутренне хранятся в двоичном формате. В частности, разряды вещественного числа представляются через элементы числового ряда  $n = \frac{x}{2} + \frac{y}{4} + \frac{z}{8} + \dots$ . Для некоторых вещественных чисел такой ряд может включать очень большое или даже бесконечное количество элементов. Но поскольку разрядность числа (количество битов, отведенных для его хранения) так или иначе ограничена, возникает неизбежная погрешность округления. Остальные числа могут быть представлены точно, поэтому даже при конечной разрядности они всегда воспроизводятся правильно. Рассмотрим следующий пример.

```
In [13]: c = 0.5
          c.as_integer_ratio()
Out[13]: (1, 2)
```

Число `0.5` хранится в исходном виде, поскольку его числовой ряд имеет конечную длину:  $0.5 = 1/2$ . А вот представление объекта `b = 0.35` совсем не такое, как можно было бы предположить исходя из простейшей записи рационального числа ( $0.35 = 7/20$ ).

```
In [14]: b.as_integer_ratio()
Out[14]: (3152519739159347, 9007199254740992)
```

Точность представления числа зависит от количества битов, выделяемых для его хранения. На всех платформах, на которых выполняется Python,

поддерживается стандарт IEEE 754, описывающий формат двойной точности ([https://ru.wikipedia.org/wiki/Число\\_двойной\\_точности](https://ru.wikipedia.org/wiki/Число_двойной_точности)), в котором внутреннее представление числа имеет разрядность 64 бит, что соответствует точности до 15-го знака после запятой.

Все это имеет существенное значение в контексте финансовых вычислений, где зачастую требуется обеспечить точное или, по крайней мере, наилучшее из возможных представление чисел. Например, при суммировании большого массива чисел даже мизерная погрешность округления может в конечном итоге привести к существенному отклонению от правильного результата.

Модуль `decimal` предоставляет в распоряжение программиста объект, предназначенный для хранения вещественных чисел произвольной точности, и ряд параметров, позволяющих избежать проблем с точностью представления таких чисел.

```
In [15]: import decimal
         from decimal import Decimal

In [16]: decimal.getcontext()
Out[16]: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999,
                Emax=999999, capitals=1, clamp=0, flags=[],
                traps=[InvalidOperation, DivisionByZero, Overflow])

In [17]: d = Decimal(1) / Decimal(11)
         d
Out[17]: Decimal('0.09090909090909090909090909091')
```

Чтобы задать требуемую точность округления, необходимо поменять соответствующий атрибут объекта `Context`.

```
In [18]: decimal.getcontext().prec = 4 ❶

In [19]: e = Decimal(1) / Decimal(11)
         e
Out[19]: Decimal('0.09091')

In [20]: decimal.getcontext().prec = 50 ❷

In [21]: f = Decimal(1) / Decimal(11)
         f
Out[21]: Decimal('0.090909090909090909090909090909090909090909091')
```



```
'except',  
'finally',  
'for',  
'from',  
'global',  
'if',  
'import',  
'in',  
'is',  
'lambda',  
'nonlocal',  
'not',  
'or',  
'pass',  
'raise',  
'return',  
'try',  
'while',  
'with',  
'yield']
```

Значения `True` и `False` относятся к типу данных `bool` и называются *булевыми*. В следующем коде выполняются различные операции *сравнения* над одними и теми же операндами.

```
In [25]: 4 > 3 ❶  
Out[25]: True
```

```
In [26]: type(4 > 3)  
Out[26]: bool
```

```
In [27]: type(False)  
Out[27]: bool
```

```
In [28]: 4 >= 3 ❷  
Out[28]: True
```

```
In [29]: 4 < 3 ❸  
Out[29]: False
```

```
In [30]: 4 <= 3 ❹  
Out[30]: False
```

```
In [31]: 4 == 3 ❺
```

```
Out[31]: False
```

```
In [32]: 4 != 3 ⑥
```

```
Out[32]: True
```

- ① Больше.
- ② Больше или равно.
- ③ Меньше.
- ④ Меньше или равно.
- ⑤ Равно.
- ⑥ Не равно.

Очень часто над булевыми значениями выполняются *логические операции*, результатом которых тоже будет объект `bool`.

```
In [33]: True and True
```

```
Out[33]: True
```

```
In [34]: True and False
```

```
Out[34]: False
```

```
In [35]: False and False
```

```
Out[35]: False
```

```
In [36]: True or True
```

```
Out[36]: True
```

```
In [37]: True or False
```

```
Out[37]: True
```

```
In [38]: False or False
```

```
Out[38]: False
```

```
In [39]: not True
```

```
Out[39]: False
```

```
In [40]: not False
```

```
Out[40]: True
```

Логические операторы и операторы сравнения разрешается свободно комбинировать.

```
In [41]: (4 > 3) and (2 > 3)
```

```
Out[41]: False
```

```
In [42]: (4 == 3) or (2 != 3)
```

```
Out[42]: True
```

```
In [43]: not (4 != 4)
```

```
Out[43]: True
```

```
In [44]: (not (4 != 4)) and (2 == 3)
```

```
Out[44]: False
```

Чаще всего логические выражения применяются в условных конструкциях, определяющих порядок выполнения программы.

```
In [45]: if 4 > 3: ❶  
        print('Условие истинно') ❷  
        Условие истинно
```

```
In [46]: i = 0 ❸  
        while i < 4: ❹  
            print('Условие истинно, i = ', i) ❺  
            i += 1 ❻  
        Условие истинно, i = 0  
        Условие истинно, i = 1  
        Условие истинно, i = 2  
        Условие истинно, i = 3
```

- ❶ Если условие истинно, то выполняется следующий код.
- ❷ Код, который выполняется в случае истинности условия.
- ❸ Инициализация переменной `i` значением 0.
- ❹ Следующий код выполняется (многократно) до тех пор, пока условие истинно.
- ❺ Вывод текста и значения переменной `i`.
- ❻ Увеличение переменной `i` на единицу; выражение `i += 1` равнозначно выражению `i = i + 1`.

В числовых выражениях булево значение `True` заменяется на 1, а значение `False` — на 0. Но когда числа приводятся к типу `bool` с помощью функции `bool()`, число 0 преобразуется в `False`, а все остальные числа — в `True`.



```
In [47]: int(True)
Out[47]: 1

In [48]: int(False)
Out[48]: 0

In [49]: float(True)
Out[49]: 1.0

In [50]: float(False)
Out[50]: 0.0

In [51]: bool(0)
Out[51]: False

In [52]: bool(0.0)
Out[52]: False

In [53]: bool(1)
Out[53]: True

In [54]: bool(10.5)
Out[54]: True

In [55]: bool(-2)
Out[55]: True
```

## Строки

После знакомства с числовыми типами настало время поговорить о строковых значениях. В Python им соответствует тип `str`. Объект `str` обладает целым рядом встроенных методов, что является одним из преимуществ Python в плане обработки строковых данных и текстовых файлов. Строковое значение берется в одинарные или двойные кавычки. Также можно преобразовать другой объект в строку с помощью функции `str()` (она использует либо стандартное представление объекта, либо представление, заданное пользователем).

```
In [56]: t = 'this is a string object'
```

С помощью встроенных методов можно, например, сделать первую букву строки прописной.

```
In [57]: t.capitalize()
Out[57]: 'This is a string object'
```

Также можно разбить строку на слова, получив список слов (объекты `list` будут рассматриваться далее).

```
In [58]: t.split()
Out[58]: ['this', 'is', 'a', 'string', 'object']
```

Можно выполнить поиск слова в строке и определить позицию (индекс) первой буквы слова в строке.

```
In [59]: t.find('string')
Out[59]: 10
```

Если слово не найдено в строке, метод `find()` вернет значение `-1`.

```
In [60]: t.find('Python')
Out[60]: -1
```

Для замены символов в строке применяется метод `replace()`.

```
In [61]: t.replace(' ', '|')
Out[61]: 'this|is|a|string|object'
```

Другой полезный метод — `strip()`, позволяющий удалять начальные или конечные символы строки.

```
In [62]: 'http://www.python.org'.strip('http://')
Out[62]: 'www.python.org'
```

Наиболее часто применяемые строковые методы Python перечислены в табл. 3.1.

*Таблица 3.1. Полезные методы для работы со строками*

Метод	Аргументы	Результат
<code>capitalize()</code>	—	Создает копию строки, в которой первая буква прописная
<code>count()</code>	<code>sub[, start[, end]]</code>	Подсчитывает число вхождений подстроки <code>sub</code>
<code>encode()</code>	<code>[encoding[, errors]]</code>	Возвращает закодированную версию строки
<code>find()</code>	<code>sub[, start[, end]]</code>	Возвращает позицию первого вхождения подстроки <code>sub</code>
<code>join()</code>	<code>seq</code>	Выполняет конкатенацию строки в последовательности <code>seq</code>
<code>replace()</code>	<code>old, new[, count]</code>	Заменяет подстроку <code>old</code> подстрокой <code>new</code> указанное число раз ( <code>count</code> )
<code>split()</code>	<code>[sep[, maxsplit]]</code>	Возвращает список слов строки, используя символ <code>sep</code> в качестве разделителя

Метод	Аргументы	Результат
<code>splitlines()</code>	<code>[keepends]</code>	Разбивает строку на подстроки в тех местах, где встречаются символы конца строки, оставляя сами символы, если параметр <i>keepends</i> равен True
<code>strip()</code>	<i>chars</i>	Создает копию строки, из которой удалены все начальные и конечные символы <i>chars</i>
<code>upper()</code>	—	Возвращает копию строки, в которой все символы переведены в верхний регистр



### Строки в кодировке Unicode

Ключевое отличие версии Python 3.7 от Python 2.7 заключается в возможности кодирования и декодирования строк, а также в поддержке стандарта Unicode (<http://bit.ly/1x41ytu>). Мы не будем углубляться в эту тему, поскольку в книге нам предстоит работать преимущественно с числовыми данными, а используемые текстовые строки будут содержать стандартные символы.

## Пример: вывод и замена строк

Для вывода на экран строк или строковых представлений объектов в Python применяется функция `print()`.

```
In [63]: print('Python for Finance') ❶
Python for Finance
```

```
In [64]: print(t) ❷
this is a string object
```

```
In [65]: i = 0
while i < 4:
    print(i) ❸
    i += 1
0
1
2
3
```

```
In [66]: i = 0
while i < 4:
    print(i, end='|') ❹
```

```
i += 1  
0|1|2|3|
```

- ❶ Выводит на экран заданную строку.
- ❷ Выводит на экран значение строковой переменной.
- ❸ Выводит на экран строковое представление целого числа (объекта `int`).
- ❹ Добавляет в конец выводимой строки завершающий символ (по умолчанию используется символ конца строки `\n`).

В Python поддерживаются удобные операции замены строк. Старый способ предполагает использование оператора `%`. Новый способ основан на использовании фигурных скобок (`{}`) и метода `format()`. На практике применяются оба способа. Мы не будем углубляться в детали, рассмотрев только самые важные варианты. Сначала приведем примеры замены строк старым способом.

```
In [67]: 'Это целое %d' % 15 ❶  
Out[67]: 'Это целое 15'
```

```
In [68]: 'Это целое %4d' % 15 ❷  
Out[68]: 'Это целое   15'
```

```
In [69]: 'Это целое %04d' % 15 ❸  
Out[69]: 'Это целое 0015'
```

```
In [70]: 'Это вещественное %f' % 15.3456 ❹  
Out[70]: 'Это вещественное 15.345600'
```

```
In [71]: 'Это вещественное %.2f' % 15.3456 ❺  
Out[71]: 'Это вещественное 15.35'
```

```
In [72]: 'Это вещественное %8f' % 15.3456 ❻  
Out[72]: 'Это вещественное 15.345600'
```

```
In [73]: 'Это вещественное %8.2f' % 15.3456 ❼  
Out[73]: 'Это вещественное   15.35'
```

```
In [74]: 'Это вещественное %08.2f' % 15.3456 ❽  
Out[74]: 'Это вещественное 00015.35'
```

```
In [75]: 'Это строка %s' % 'Python' ❾  
Out[75]: 'Это строка Python'
```

```
In [76]: 'Это строка %10s' % 'Python' ⑩  
Out[76]: 'Это строка      Python'
```

- ① Замена объектом `int`.
- ② Замена объектом `int` с фиксированным количеством символов.
- ③ Замена объектом `int` с дополнением ведущими нулями в случае необходимости.
- ④ Замена объектом `float`.
- ⑤ Замена объектом `float` с фиксированным количеством знаков после запятой.
- ⑥ Замена объектом `float` с фиксированным количеством символов (при необходимости добавляются нулевые разряды).
- ⑦ Замена объектом `float` с фиксированным количеством символов и фиксированной точностью.
- ⑧ Замена объектом `float` с фиксированным количеством символов и фиксированной точностью (при необходимости добавляются ведущие нули).
- ⑨ Замена строкой.
- ⑩ Замена строкой с фиксированным количеством символов.

Теперь выполним те же самые операции новым способом. Обратите внимание отличие в последнем примере.

```
In [77]: 'Это целое {:d}'.format(15)  
Out[77]: 'Это целое 15'
```

```
In [78]: 'Это целое {:4d}'.format(15)  
Out[78]: 'Это целое   15'
```

```
In [79]: 'Это целое {:04d}'.format(15)  
Out[79]: 'Это целое 0015'
```

```
In [80]: 'Это вещественное {:f}'.format(15.3456)  
Out[80]: 'Это вещественное 15.345600'
```

```
In [81]: 'Это вещественное {:.2f}'.format(15.3456)  
Out[81]: 'Это вещественное 15.35'
```

```
In [82]: 'Это вещественное {:.8f}'.format(15.3456)
```

```

Out[82]: 'Это вещественное 15.345600'

In [83]: 'Это вещественное {:.2f}'.format(15.3456)
Out[83]: 'Это вещественное    15.35'

In [84]: 'Это вещественное {:08.2f}'.format(15.3456)
Out[84]: 'Это вещественное 00015.35'

In [85]: 'Это строка {:s}'.format('Python')
Out[85]: 'Это строка Python'

In [86]: 'Это строка {:10s}'.format('Python')
Out[86]: 'Это строка Python      '

```

Замена строк часто применяется при многократном выводе меняющихся данных, например в цикле `while`.

```

In [87]: i = 0
        while i < 4:
            print('Число равно %d' % i)
            i += 1
        Число равно 0
        Число равно 1
        Число равно 2
        Число равно 3

In [88]: i = 0
        while i < 4:
            print('the number is {:d}'.format(i))
            i += 1
        Число равно 0
        Число равно 1
        Число равно 2
        Число равно 3

```

## Пример: регулярные выражения

Для работы со строковыми объектами часто применяются *регулярные выражения*. Соответствующие функции содержатся в модуле `re`.

```
In [89]: import re
```

Представьте себя на месте финансового аналитика, который получил большой текстовый файл, например CSV-файл, содержащий временные ряды. Чаще всего информация о дате/времени представляется в формате, с которым

код Python не может работать напрямую. Для преобразования такой информации в правильный формат нужны регулярные выражения. В качестве примера рассмотрим строковый объект, содержащий три элемента даты/времени, три целочисленных значения и три строки. Обратите внимание на то, что значение объекта заключено в тройные кавычки. Это позволяет разбивать запись на несколько строк.

```
In [90]: series = """
'01/18/2014 13:00:00', 100, '1st';
'01/18/2014 13:30:00', 110, '2nd';
'01/18/2014 14:00:00', 120, '3rd'
"""
```

Следующее регулярное выражение описывает применяемый формат даты/времени<sup>4</sup>.

```
In [91]: dt = re.compile("[0-9/:\s]+") # дата и время
```

Воспользовавшись им, вы легко найдете все значения даты/времени. В такого рода задачах поиск с помощью регулярных выражений выполняется быстрее, чем с помощью стандартных средств.

```
In [92]: result = dt.findall(series)
result
Out[92]: ["'01/18/2014 13:00:00'", "'01/18/2014 13:30:00'",
"'01/18/2014 14:00:00'"]
```



### Регулярные выражения

Сложные операции поиска подстрок в строковых объектах лучше выполнять с помощью регулярных выражений, которые обеспечивают преимущества с точки зрения производительности.

Все найденные строковые значения даты/времени можно преобразовать в объекты специального типа — `datetime` (подробнее эти операции рассмотрены в приложении А). Для этого необходимо указать структуру хранимого значения.

```
In [93]: from datetime import datetime
pydt = datetime.strptime(result[0].replace("'", ""),
                          '%m/%d/%Y %H:%M:%S')
pydt
```

---

<sup>4</sup> Здесь мы не будем углубляться в детали таких выражений. В Интернете можно найти множество информации о регулярных выражениях в целом и их синтаксисе в Python в частности. Хорошее введение в тему регулярных выражений содержится в книге Фицджеральда [4].

```
Out[93]: datetime.datetime(2014, 1, 18, 13, 0)
```

```
In [94]: print(pydt)
2014-01-18 13:00:00
```

```
In [95]: print(type(pydt))
<class 'datetime.datetime'>
```

Подробнее о работе со значениями даты и времени, а также о применяемых для этого объектах `datetime` и их методах мы поговорим в следующих главах. А пока что лишь обозначим важность этой темы в контексте финансовых расчетов.

## Основные структуры данных

Структуры данных — это объекты, которые могут содержать произвольное число других объектов. В Python поддерживаются следующие встроенные структуры данных.

### `tuple`

Неизменяемый набор произвольных объектов с поддержкой всего нескольких методов.

### `list`

Изменяемый набор произвольных объектов с поддержкой большого количества методов.

### `dict`

Набор объектов, доступ к которым осуществляется по ключу.

### `set`

Неупорядоченный набор *уникальных* объектов.

## Кортежи

*Кортеж* (объект `tuple`) — достаточно простая структура ограниченного применения. Элементы кортежа перечисляются через запятую и заключаются в круглые скобки.

```
In [96]: t = (1, 2.5, 'data')
         type(t)
Out[96]: tuple
```



Допускается не указывать круглые скобки, перечисляя только сами объекты через запятую.

```
In [97]: t = 1, 2.5, 'data'
         type(t)
Out[97]: tuple
```

Элементы кортежа, как и в случае остальных структур данных, имеют встроенную индексацию, что позволяет обращаться к ним по отдельности. Важно помнить о том, что в Python индексация ведется с нуля, а не с единицы. Таким образом, третий элемент кортежа будет иметь индекс 2.

```
In [98]: t[2]
Out[98]: 'data'
```

```
In [99]: type(t[2])
Out[99]: str
```



### Индексация с нуля

В отличие от некоторых других языков программирования, например Matlab, в Python применяется индексация с нуля. Это означает, что первый элемент кортежа всегда имеет индекс 0.

У объекта `tuple` всего два метода: `count()` и `index()`. Первый возвращает количество экземпляров заданного объекта в кортеже, а второй возвращает индекс первого из них.

```
In [100]: t.count('data')
Out[100]: 1
```

```
In [101]: t.index(1)
Out[101]: 0
```

Кортежи — *неизменяемые* объекты. Это означает, что созданный объект не подлежит модификации.

## Списки

*Список* (объект `list`) — более гибкая и функциональная структура данных, чем кортеж. Списки широко применяются в финансовых вычислениях, например для хранения регулярно обновляемых котировок акций. Элементы списка перечисляются через запятую и заключаются в квадратные скобки. Базовая функциональность списков напоминает функциональность кортежей.

```
In [102]: l = [1, 2.5, 'data']
          l[2]
Out[102]: 'data'
```

Списки можно также создавать с помощью функции `list()`. В следующем примере созданный ранее кортеж преобразуется в список.

```
In [103]: l = list(t)
          l
Out[103]: [1, 2.5, 'data']
```

```
In [104]: type(l)
Out[104]: list
```

В отличие от кортежей и строк списки можно расширять и сокращать с помощью различных методов и операций, поскольку это *изменяемые* объекты. В частности, можно присоединить список к существующему списку.

```
In [105]: l.append([4, 3]) ❶
          l
Out[105]: [1, 2.5, 'data', [4, 3]]
```

```
In [106]: l.extend([1.0, 1.5, 2.0]) ❷
          l
Out[106]: [1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]
```

```
In [107]: l.insert(1, 'insert') ❸
          l
Out[107]: [1, 'insert', 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]
```

```
In [108]: l.remove('data') ❹
          l
Out[108]: [1, 'insert', 2.5, [4, 3], 1.0, 1.5, 2.0]
```

```
In [109]: p = l.pop(3) ❺
          print(l, p)
          [1, 'insert', 2.5, 1.0, 1.5, 2.0] [4, 3]
```

- ❶ Добавление списка в конец другого списка.
- ❷ Добавление элементов в список.
- ❸ Вставка объекта в указанную позицию списка.
- ❹ Удаление первого экземпляра объекта из списка.

- ⑤ Удаление объекта из списка по указанной позиции; метод возвращает извлеченный объект.

При работе со списками можно создавать *срезы*, т.е. фрагменты исходного списка.

```
In [110]: l[2:5] ①
Out[110]: [2.5, 1.0, 1.5]
```

- ① Срез включает с третьего по пятый элементы исходного списка.

Наиболее часто используемые операции и методы списков перечислены в табл. 3.2.

Таблицы 3.2. Методы объекта *list* и операции со списками

Метод	Аргументы	Описание
<code>l[i] = x</code>	<code>[i]</code>	Заменяет <i>i</i> -й элемент объектом <i>x</i>
<code>l[i:j:k] = s</code>	<code>[i:j:k]</code>	Заменяет каждый <i>k</i> -й элемент в диапазоне от <i>i</i> до <i>j</i> - 1 объектом <i>s</i>
<code>append</code>	<code>(x)</code>	Добавляет объект <i>x</i> в конец списка
<code>count</code>	<code>(x)</code>	Посчитывает количество экземпляров объекта <i>x</i>
<code>del l[i:j:k]</code>	<code>[i:j:k]</code>	Удаляет каждый <i>k</i> -й элемент в диапазоне от <i>i</i> до <i>j</i> - 1
<code>extend</code>	<code>(s)</code>	Включает в список все элементы объекта <i>s</i>
<code>index</code>	<code>(x[, i[, j]])</code>	Определяет индекс первого экземпляра <i>x</i> , встречающегося в диапазоне от <i>i</i> до <i>j</i> - 1
<code>insert</code>	<code>(i, x)</code>	Вставляет объект <i>x</i> перед позицией с индексом <i>i</i>
<code>remove</code>	<code>(x)</code>	Удаляет первое вхождение объекта <i>x</i>
<code>pop</code>	<code>(i)</code>	Удаляет элемент с индексом <i>i</i> и возвращает его
<code>reverse</code>	—	Изменяет порядок элементов на обратный
<code>sort</code>	<code>([cmp[, key[, reverse]]])</code>	Сортирует элементы списка указанным способом

Пример: управляющие конструкции

Несмотря на то что это отдельная тема, знакомство с *управляющими конструкциями* Python, такими как цикл `for`, лучше всего начинать в контексте списков. Дело в том, что, в отличие от других языков программирования, в Python циклы применяются по отношению к объектам *list*. Рассмотрим пример, в котором цикл `for` применяется для извлечения элементов списка `l` с

индексами от 2 до 4, возведения их в квадрат и вывода полученных значений на экран. Обратите внимание на отступ второй строки — он необходим для структурирования кода и означает, что данная строка выполняется в цикле `for`.

```
In [111]: for element in l[2:5]:
           print(element ** 2)
6.25
1.0
2.25
```

Это более гибкий подход по сравнению со стандартными циклами, основанными на использовании счетчиков. Последние тоже поддерживаются в Python, но реализуются с помощью объекта `range`.

```
In [112]: r = range(0, 8, 1) ❶
r
Out[112]: range(0, 8)
```

```
In [113]: type(r)
Out[113]: range
```

- ❶ Параметрами конструктора являются начальный элемент диапазона, конечный элемент диапазона и шаг приращения.

Для сравнения создадим аналогичный цикл с использованием объекта `range`.

```
In [114]: for i in range(2, 5):
           print(l[i] ** 2)
6.25
1.0
2.25
```



#### Просмотр элементов списка

В Python можно циклически перебирать элементы произвольных списков, что избавляет от необходимости использовать счетчики.

В Python имеются и типичные условные конструкции `if`, `elif` и `else`, которые применяются так же, как и в других языках программирования.

```
In [115]: for i in range(1, 10):
           if i % 2 == 0: ❶
               print("%d четное" % i)
           elif i % 3 == 0:
               print("%d кратно 3" % i)
```

```

        else:
            print("%d нечетное" % i)
1 нечетное
2 четное
3 кратно 3
4 четное
5 нечетное
6 четное
7 нечетное
8 четное
9 кратно 3

```

❶ Здесь % — это оператор деления по модулю.

Циклическую обработку элементов можно реализовать и в цикле `while`.

```

In [116]: total = 0
         while total < 100:
             total += 1
         print(total)
         100

```

Особенностью Python являются так называемые *списковые включения*, которые позволяют компактно генерировать списки в циклах, вместо того чтобы циклически перебирать существующие списки.

```

In [117]: m = [i ** 2 for i in range(5)]
           m
Out[117]: [0, 1, 4, 9, 16]

```

В определенном смысле подобные списковые включения, в которых код цикла задан фактически в неявном виде, можно считать первым шагом на пути к векторизации вычислений (о векторизации кода мы поговорим в главах 4 и 5).

## Пример: функциональное программирование

Python включает целый ряд средств функционального программирования, например функции `filter()`, `map()` и `reduce()`, которые можно применять ко всему набору входных данных (в нашем случае — ко всему содержимому объекта `list`). Но сначала необходимо разобраться с тем, как объявляется функция. Создадим простейшую функцию `f()`, возвращающую квадрат передаваемого ей аргумента `x`.

```
In [118]: def f(x):  
          return x ** 2  
          f(2)  
Out[118]: 4
```

Реальные функции могут быть сколь угодно сложными, с множеством аргументов и вариантов возвращаемых значений. Рассмотрим следующую функцию.

```
In [119]: def even(x):  
          return x % 2 == 0  
          even(3)  
Out[119]: False
```

Она возвращает булево значение. Такую функцию можно применить сразу ко всему списку с помощью функции `map()`.

```
In [120]: list(map(even, range(10)))  
Out[120]: [True, False, True, False, True, False, True, False,  
           True, False]
```

Более того, в качестве аргумента функции `map()` можно передать определение анонимной функции с помощью ключевого слова `lambda`. Такие анонимные функции называются *лямбда-функциями*.

```
In [121]: list(map(lambda x: x ** 2, range(10)))  
Out[121]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

С помощью функций можно фильтровать списки. В следующем примере функция `filter()` возвращает элементы списка, которые соответствуют булевому условию, заданному в функции `even()`.

```
In [122]: list(filter(even, range(15)))  
Out[122]: [0, 2, 4, 6, 8, 10, 12, 14]
```



### Списковые включения, функциональное программирование и анонимные функции

В Python считается хорошей практикой всячески избегать использования циклов. В этом смысле списковые включения и средства функционального программирования, в частности функции `filter()`, `map()` и `reduce()`, помогают писать более компактный и понятный код без (явных) циклов. Лямбда-функции служат тем же самым целям.

## Словари

*Словари* (объекты `dict`) — это изменяемые наборы объектов, индексируемые специальными ключами (чаще всего строками). В отличие от списков, словари не упорядочены и не сортируются<sup>5</sup>. Элементы словаря перечисляются через запятую и заключаются в фигурные скобки. Следующий пример иллюстрирует разницу между списками и словарями.

```
In [123]: d = {
            'Имя' : 'Ангела Меркель',
            'Страна' : 'Германия',
            'Должность' : 'Канцлер',
            'Возраст' : 64
        }
        type(d)
Out[123]: dict
```

```
In [124]: print(d['Имя'], d['Возраст'])
Ангела Меркель 64
```

У словарей имеется множество встроенных методов.

```
In [125]: d.keys()
Out[125]: dict_keys(['Имя', 'Страна', 'Должность', 'Возраст'])

In [126]: d.values()
Out[126]: dict_values(['Ангела Меркель', 'Германия', 'Канцлер', 64])

In [127]: d.items()
Out[127]: dict_items([('Имя', 'Ангела Меркель'), ('Страна',
            'Германия'), ('Должность', 'Канцлер'),
            ('Возраст', 64)])

In [128]: birthday = True
            if birthday:
                d['Возраст'] += 1
            print(d['Возраст'])
65
```

На основе словаря можно получить объект `iterator`, который применяется в циклах подобно спискам.

---

<sup>5</sup> Существует подкласс словарей `OrderedDict`, хранящий сведения о порядке его заполнения элементами (<https://docs.python.org/3/library/collections.html>).

```
In [129]: for item in d.items():
           print(item)
('Имя', 'Ангела Меркель')
('Страна', 'Германия')
('Должность', 'Канцлер')
('Возраст', 65)
```

```
In [130]: for value in d.values():
           print(type(value))
<class 'str'>
<class 'str'>
<class 'str'>
<class 'int'>
```

Наиболее часто используемые операции и методы словарей перечислены в табл. 3.3.

*Таблица 3.3. Методы объекта dict и операции со словарями*

Метод	Аргумент	Описание
<code>d[k]</code>	<code>[k]</code>	Получение элемента с ключом <i>k</i>
<code>d[k] = x</code>	<code>[k]</code>	Присвоение значения <i>x</i> элементу с ключом <i>k</i>
<code>del d[k]</code>	<code>[k]</code>	Удаление элемента с ключом <i>k</i>
<code>clear</code>	—	Удаление всех элементов
<code>copy</code>	—	Создание копии словаря
<code>has_key</code>	<code>(k)</code>	True, если в словаре есть ключ <i>k</i>
<code>items</code>	—	Итератор по всем элементам
<code>keys</code>	—	Итератор по всем ключам
<code>values</code>	—	Итератор по всем значениям
<code>popitem</code>	<code>(k)</code>	Возвращает элемент с ключом <i>k</i> , удаляя его из словаря
<code>update</code>	<code>([e])</code>	Обновляет элементы словаря значениями из объекта <i>e</i>

## Множества

Последняя из рассматриваемых в этой главе структур данных — *множество* (объект `set`). Несмотря на фундаментальную роль теории множеств в математике и финансах, практических применений объектов `set` не так уж много. Множество представляет собой неупорядоченный набор уникальных элементов.



```
In [131]: s = set(['u', 'd', 'ud', 'du', 'd', 'du'])
          s
Out[131]: {'d', 'du', 'u', 'ud'}
```

```
In [132]: t = set(['d', 'dd', 'uu', 'u'])
```

Над объектами `set` можно выполнять математические операции из теории множеств, такие как объединение, пересечение и вычитание.

```
In [133]: s.union(t) ❶
Out[133]: {'d', 'dd', 'du', 'u', 'ud', 'uu'}
```

```
In [134]: s.intersection(t) ❷
Out[134]: {'d', 'u'}
```

```
In [135]: s.difference(t) ❸
Out[135]: {'du', 'ud'}
```

```
In [136]: t.difference(s) ❹
Out[136]: {'dd', 'uu'}
```

```
In [137]: s.symmetric_difference(t) ❺
Out[137]: {'dd', 'du', 'ud', 'uu'}
```

- ❶ Объединение всех элементов множеств `s` и `t`.
- ❷ Элементы, встречающиеся в обоих множествах (`s` и `t`).
- ❸ Элементы множества `s`, не встречающиеся в множестве `t`.
- ❹ Элементы множества `t`, не встречающиеся в множестве `s`.
- ❺ Элементы, встречающиеся в множестве `s` или `t`, но не в обоих сразу.

Одно из применений объектов `set` — удаление дубликатов из объектов `list`.

```
In [138]: from random import randint
          l = [randint(0, 10) for i in range(1000)] ❶
          len(l) ❷
Out[138]: 1000
```

```
In [139]: l[:20]
Out[139]: [4, 2, 10, 2, 1, 10, 0, 6, 0, 8, 10, 9, 2, 4, 7, 8, 10,
           8, 8, 2]
```

```
In [140]: s = set(l)
```

```
s
```

```
Out[140]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

- ❶ 1000 случайных целых чисел в диапазоне от 0 до 10.
- ❷ Количество элементов в списке `l`.

## Резюме

В Python поддерживается богатый набор средств для работы с данными, которые мы будем применять в финансовых расчетах. В этой главе мы рассмотрели следующие встроенные средства языка.

### *Основные типы данных*

В Python в целом и в финансовых расчетах в частности фундаментальными типами данных являются `int`, `float`, `bool` и `str`.

### *Основные структуры данных*

В Python поддерживаются структуры типа `tuple`, `list`, `dict` и `set`, среди которых самая гибкая и универсальная — список (объект `list`).

## Дополнительные ресурсы

Рассматривая основные типы и структуры данных, мы делали акцент на том, что может быть важно с точки зрения финансовых приложений. Главу можно считать лишь введением в тему структур данных на Python.

Дополнительную информацию можно получить из целого ряда источников. Первый из них — это, конечно же, официальная документация:

<https://docs.python.org/3/tutorial/datastructures.html>

Также хорошими источниками послужат следующие издания.

1. Goodrich, Michael, et al. *Data Structures and Algorithms in Python* (2013, Wiley).
2. Harrison, Matt. *Illustrated Guide to Python 3* (2017, CreateSpace).
3. Ramalho, Luciano. *Fluent Python* (2016, O'Reilly).
4. Фицджеральд, Майкл. *Регулярные выражения: основы* (пер. с англ., ИД “Вильямс”, 2015).



# Работа с массивами NumPy

Компьютеры бесполезны. Они могут только давать ответы.

*Пабло Пикассо*

Несмотря на то что в Python имеется большое количество базовых структур данных, пользователям предлагаются еще более широкие возможности, реализуемые в дополнительных библиотеках. В этой главе мы познакомимся с библиотекой NumPy, в которой содержится класс многомерных массивов, позволяющий обрабатывать однородные и неоднородные структуры данных. Другое преимущество библиотеки — поддержка векторизации кода.

В главе рассматриваются следующие структуры данных.

Тип объекта	Описание	Применение
<code>ndarray</code> (обычный)	Объект <i>n</i> -мерного массива	Большие массивы числовых данных
<code>ndarray</code> (запись)	Объект двумерного массива	Табличные данные с разбивкой по столбцам

Общая структура главы такова.

### Массивы данных

Этот раздел посвящен работе с массивами с использованием встроенных средств Python.

### Обычные массивы NumPy

В этом разделе рассматривается класс `ndarray` библиотеки NumPy, который является основным инструментом работы с числовыми массивами в Python.

### Структурированные массивы NumPy

Этот раздел включает короткое описание структурированных объектов `ndarray` (*записей*), предназначенных для хранения табличных данных с разбивкой по столбцам.

### Векторизация кода

В этом разделе описываются преимущества векторных операций с массивами.

# Массивы данных

В предыдущей главе вы познакомились с базовыми структурами данных, имеющимися в Python. Чаще всего мы будем иметь дело с объектами `list`, которые благодаря большой гибкости находят широкое применение. Но у гибкости есть своя цена, которая проявляется в относительно высоких требованиях к памяти, а также в снижении производительности. В научных и финансовых приложениях фактор скорости вычислений очень важен, поэтому приходится работать со специальными структурами данных, самая важная из которых — *массив*. В массивах хранятся объекты *одного и того же типа* с разбивкой по строкам и столбцам.

Возьмем числовые массивы. Самый простой случай — одномерный массив, который с математической точки зрения представляет собой *вектор*, состоящий из одной строки или одного столбца чисел. В более общем случае массив представляет собой *матрицу* размером  $i \times j$ . Следующий уровень — *куб* в трехмерном пространстве размером  $i \times j \times k$ . Далее концепция обобщается на  $n$ -мерный массив размером  $i \times j \times k \times l \times \dots$

Из курса высшей математики вы должны знать, что такого рода математические структуры играют важную роль во многих областях науки. Поэтому чрезвычайно полезно иметь специальный класс, предназначенный для эффективной работы с массивами. Вот здесь нам и пригодится библиотека NumPy, включающая удобный класс `ndarray`. Но для начала следует рассмотреть два альтернативных способа обработки массивов.

## Преобразование списка в массив

Массив можно образовать из любой встроенной структуры данных, рассмотренной в предыдущей главе. Особенно удобны для этого списки (объекты `list`). Простой список уже можно считать одномерным массивом.

```
In [1]: v = [0.5, 0.75, 1.0, 1.5, 2.0] ❶
```

❶ Список чисел.

Списки могут содержать произвольные объекты, в том числе другие списки. Таким образом, матрицы и многомерные массивы можно легко создавать с помощью вложенных списков.

```
In [2]: m = [v, v, v] ❶  
        m ❷
```

```
Out[2]: [[0.5, 0.75, 1.0, 1.5, 2.0],
```

```
[0.5, 0.75, 1.0, 1.5, 2.0],  
[0.5, 0.75, 1.0, 1.5, 2.0]]
```

- ❶ Список, включающий другие списки...
- ❷ ...представляет собой матрицу.

Строки матрицы легко извлекаются с помощью одиночного индекса, а отдельные элементы — с помощью двойной индексации (столбцы матрицы выбрать подобным образом не получится).

```
In [3]: m[1]  
Out[3]: [0.5, 0.75, 1.0, 1.5, 2.0]
```

```
In [4]: m[1][0]  
Out[4]: 0.5
```

Дальнейшее вложение списков позволяет получить многомерные структуры.

```
In [5]: v1 = [0.5, 1.5]  
        v2 = [1, 2]  
        m = [v1, v2]  
        c = [m, m] ❶  
        c  
Out[5]: [[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]  
  
In [6]: c[1][1][0]  
Out[6]: 1
```

- ❶ Трехмерный куб.

При таком способе комбинирования объектов создаются ссылки на исходные объекты. К чему это может привести? Рассмотрим следующий пример.

```
In [7]: v = [0.5, 0.75, 1.0, 1.5, 2.0]  
        m = [v, v, v]  
        m  
Out[7]: [[0.5, 0.75, 1.0, 1.5, 2.0],  
         [0.5, 0.75, 1.0, 1.5, 2.0],  
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

Давайте заменим первый элемент объекта `v` и посмотрим, как изменится объект `m`.

```
In [8]: v[0] = 'Python'  
        m
```

```
Out[8]: [['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0]]
```

Подобного поведения можно избежать, применив функцию `deepcopy()` из модуля `copy`.

```
In [9]: from copy import deepcopy
        v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = 3 * [deepcopy(v), ] ❶
        m
```

```
Out[9]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

```
In [10]: v[0] = 'Python' ❷
         m ❸
```

```
Out[10]: [[0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0]]
```

- ❶ Вместо ссылки в массив добавляется копия объекта.
- ❷ В результате изменение исходного объекта...
- ❸ ...не приводит к изменению самого массива.

## Класс `array`

В стандартную библиотеку Python включен модуль `array`, содержащий одноименный класс. Вот как он описан в официальной документации (<https://docs.python.org/3/library/array.html>).

Данный модуль определяет класс, предназначенный для компактного хранения массивов простых типов: символов, целых чисел и чисел с плавающей точкой. Массивы — это структурированные типы, напоминающие списки, но в отличие от последних в них разрешено хранить значения одного типа. Тип указывается на этапе создания объекта и обозначается специальным односимвольным кодом.

Рассмотрим пример, в котором объект `array` создается на основе списка.

```
In [11]: v = [0.5, 0.775, 1.0, 1.5, 2.0]
```

```
In [12]: import array
```

```

In [13]: a = array.array('f', v) ❶
a
Out[13]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])

In [14]: a.append(0.5) ❷
a
Out[14]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5])

In [15]: a.extend([5.0, 6.75]) ❷
a
Out[15]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])

In [16]: 2 * a ❸
Out[16]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75, 0.5,
                    0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])

```

- ❶ Создание объекта `array` с типом элементов `float`.
- ❷ Методы массивов работают так же, как и в случае списков.
- ❸ Несмотря на операцию скалярного умножения, результат получается не таким, как ожидается: элементы массива просто копируются.

Попытка добавить в массив объект другого типа вызывает ошибку `TypeError`.

```

In [17]: a.append('string') ❶

-----
TypeError                                Traceback (most recent call last)
<ipython-input-17-14cd6281866b> in <module>()
----> 1 a.append('string') ❶

TypeError: must be real number, not str

```

```

In [18]: a.tolist() ❷
Out[18]: [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75]

```

- ❶ В исходный массив разрешается добавлять только объекты `float`. Во всех остальных случаях выдается сообщение об ошибке.
- ❷ В случае необходимости массив можно преобразовать обратно в список.



Преимуществом класса `аггау` является наличие встроенных методов для работы с файлами.

```
In [19]: f = open('array.apy', 'wb') ❶  
         a.tofile(f) ❷  
         f.close() ❸
```

```
In [20]: with open('array.apy', 'wb') as f: ❹  
         a.tofile(f) ❺
```

```
In [21]: !ls -n arr* ❻  
-rw-r--r--@ 1 503  20  32 Nov  7 11:46 array.apy
```

- ❶ Открытие файла на диске для записи двоичных данных.
- ❷ Запись содержимого объекта `аггау` в файл.
- ❸ Закрытие файла.
- ❹ Альтернативный синтаксис с использованием конструкции `with`.
- ❺ Просмотр информации о файле.

При считывании данных с диска необходимо учитывать их тип.

```
In [22]: b = array.array('f') ❶
```

```
In [23]: with open('array.apy', 'rb') as f: ❷  
         b.fromfile(f, 5) ❸
```

```
In [24]: b❹  
Out[24]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])
```

```
In [25]: b = array.array('d') ❺
```

```
In [26]: with open('array.apy', 'rb') as f:  
         b.fromfile(f, 2) ❻
```

```
In [27]: b ❼  
Out[27]: array('d', [0.0004882813645963324, 0.12500002956949174])
```

- ❶ Объявление нового массива с типом данных `float`.
- ❷ Открытие файла для чтения двоичных данных.
- ❸ Считывание пяти элементов в массив `b`.

- ❹ Объявление нового массива с типом данных `double`.
- ❺ Считывание двух элементов из файла.
- ❻ Изменение типа данных массива приводит к считыванию неправильного формата чисел.

## Обычные массивы NumPy

Создание массивов на основе списков — не самый рациональный подход. В конце концов, класс `list` создавался не с этой целью. У списков более широкая область применения, чем у объектов `аггау`. Полезнее было бы иметь более специализированный класс для работы с различного вида массивами.

### Основные операции

И такой класс есть: это `numpy.ndarray`. Он специально создавался для удобной и эффективной работы с многомерными массивами. Рассмотрим конкретные примеры.

```
In [28]: import numpy as np ❶

In [29]: a = np.array([0, 0.5, 1.0, 1.5, 2.0]) ❷
          a
Out[29]: array([0., 0.5, 1., 1.5, 2.])

In [30]: type(a) ❷
Out[30]: numpy.ndarray

In [31]: a = np.array(['a', 'b', 'c']) ❸
          a
Out[31]: array(['a', 'b', 'c'], dtype='<U1')

In [32]: a = np.arange(2, 20, 2) ❹
          a
Out[32]: array([2, 4, 6, 8, 10, 12, 14, 16, 18])

In [33]: a = np.arange(8, dtype=np.float) ❺
          a
Out[33]: array([0., 1., 2., 3., 4., 5., 6., 7.])

In [34]: a[5:] ❻
```

```
Out[34]: array([5., 6., 7.])
```

```
In [35]: a[:2] ❸
```

```
Out[35]: array([0., 1.])
```

- ❶ Импорт пакета numpy.
- ❷ Создание объекта ndarray на основе списка чисел типа float.
- ❸ Создание объекта ndarray на основе списка строк.
- ❹ Метод `np.arange()` аналогичен методу `range()`...
- ❺ ...но поддерживает дополнительный аргумент `dtype`.
- ❻ Одномерные массивы ndarray индексируются привычным образом.

Отличительная особенность класса ndarray — поддержка большого количества встроенных методов.

```
In [36]: a.sum() ❶
```

```
Out[36]: 28.0
```

```
In [37]: a.std() ❷
```

```
Out[37]: 2.29128784747792
```

```
In [38]: a.cumsum() ❸
```

```
Out[38]: array([0., 1., 3., 6., 10., 15., 21., 28.])
```

- ❶ Сумма элементов массива.
- ❷ Среднеквадратическое отклонение элементов массива.
- ❸ Накопительная сумма элементов массива (начиная с позиции 0).

Другая особенность пакета NumPy — поддержка (векторизованных) *математических операций* с массивами ndarray.

```
In [39]: l = [0., 0.5, 1.5, 3., 5.]
```

```
2 * l ❶
```

```
Out[39]: [0.0, 0.5, 1.5, 3.0, 5.0, 0.0, 0.5, 1.5, 3.0, 5.0]
```

```
In [40]: a
```

```
Out[40]: array([0., 1., 2., 3., 4., 5., 6., 7.])
```

```
In [41]: 2 * a ❷
```

```
Out[41]: array([ 0., 2., 4., 6., 8., 10., 12., 14.])
```

```
In [42]: a ** 2 ❸  
Out[42]: array([0., 1., 4., 9., 16., 25., 36., 49.])
```

```
In [43]: 2 ** a ❹  
Out[43]: array([1., 2., 4., 8., 16., 32., 64., 128.])
```

```
In [44]: a ** a ❺  
Out[44]: array([1.000000e+00, 1.000000e+00, 4.000000e+00, 2.700000e+01,  
                2.560000e+02, 3.125000e+03, 4.665600e+04, 8.23543e+05])
```

- ❶ Скалярное умножение списка сводится к дублированию его элементов.
- ❷ В случае объекта `ndarray` скалярное умножение выполняется над элементами массива.
- ❸ Поэлементное возведение значений в квадрат.
- ❹ Возведение числа в степень каждого элемента массива.
- ❺ Возведение каждого элемента массива в степень своего значения.

Следующая важная особенность пакета NumPy заключается в поддержке *универсальных функций*, способных выполнять действия не только с массивами `ndarray`, но и с любыми базовыми типами данных Python. Только следует учитывать, что скорость выполнения операций, к примеру, над объектами `float` будет ниже, чем при использовании аналогичных функций модуля `math`.

```
In [45]: np.exp(a) ❶  
Out[45]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00,  
                2.00855369e+01, 5.45981500e+01, 1.48413159e+02,  
                4.03428793e+02, 1.09663316e+03])
```

```
In [46]: np.sqrt(a) ❷  
Out[46]: array([0.          , 1.          , 1.41421356, 1.73205081,  
                2.          , 2.23606798, 2.44948974, 2.64575131])
```

```
In [47]: np.sqrt(2.5) ❸  
Out[47]: 1.5811388300841898
```

```
In [48]: import math ❹
```

```
In [49]: math.sqrt(2.5) ❺  
Out[49]: 1.5811388300841898
```

In [50]: `math.sqrt(a)` ⑤

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-50-b39de4150838> in <module>()  
----> 1 math.sqrt(a)  
  
TypeError: only size-1 arrays can be converted to Python  
scalars
```

In [51]: `%timeit np.sqrt(2.5)` ⑥

722 ns  $\pm$  13.7 ns per loop (mean  $\pm$  std. dev. of 7 runs,  
1000000 loops each)

In [52]: `%timeit math.sqrt(2.5)` ⑦

91.8 ns  $\pm$  4.13 ns per loop (mean  $\pm$  std. dev. of 7 runs,  
10000000 loops each)

- ① Вычисление экспоненты каждого элемента массива.
- ② Вычисление квадратного корня каждого элемента массива.
- ③ Вычисление квадратного корня обычного объекта `float`.
- ④ Вычисляем то же самое с помощью модуля `math`.
- ⑤ Функцию `math.sqrt()` невозможно применить к объекту `ndarray` напрямую.
- ⑥ Универсальная функция `np.sqrt()` обрабатывает объекты `float`...
- ⑦ ...намного медленнее, чем функция `math.sqrt()`.

## Многомерные массивы

Переход к многомерным массивам NumPy не вызывает никаких затруднений, поскольку вся рассмотренная ранее функциональность без проблем обобщается на произвольное количество измерений.

In [53]: `b = np.array([a, a * 2])` ①  
b

Out[53]: `array([[0., 1., 2., 3., 4., 5., 6., 7.],  
[0., 2., 4., 6., 8., 10., 12., 14.]])`

In [54]: `b[0]` ②

```
Out[54]: array([0., 1., 2., 3., 4., 5., 6., 7.])
```

```
In [55]: b[0, 2] ❸
```

```
Out[55]: 2.0
```

```
In [56]: b[:, 1] ❹
```

```
Out[56]: array([1., 2.])
```

```
In [57]: b.sum() ❺
```

```
Out[57]: 84.0
```

```
In [58]: b.sum(axis=0) ❻
```

```
Out[58]: array([0., 3., 6., 9., 12., 15., 18., 21.])
```

```
In [59]: b.sum(axis=1) ❼
```

```
Out[59]: array([28., 56.])
```

- ❶ Построение двумерного массива `ndarray` на основе одномерного.
- ❷ Выбор первой строки.
- ❸ Выбор третьего элемента первой строки (индексы в квадратных скобках разделяются запятыми).
- ❹ Выбор второго столбца.
- ❺ Вычисление суммы *всех* элементов массива.
- ❻ Вычисление суммы элементов по первому измерению, т.е. по столбцам.
- ❼ Вычисление суммы элементов по второму измерению, т.е. построчно.

Существует несколько способов инициализации объекта `ndarray`. Один из них, рассмотренный выше, заключается в использовании метода `np.array()`. Но это предполагает, что все элементы массива известны заранее. Другой вариант заключается в создании пустого массива, заполняемого в процессе выполнения программы. Ниже показано, какие методы для этого применяются.

```
In [60]: c = np.zeros((2, 3), dtype='i', order='C') ❶
```

c

```
Out[60]: array([[0, 0, 0],  
               [0, 0, 0]], dtype=int32)
```

```
In [61]: c = np.ones((2, 3, 4), dtype='i', order='C') ❷
```

c

```

Out[61]: array([[[1, 1, 1, 1],
                 [1, 1, 1, 1],
                 [1, 1, 1, 1]],

               [[1, 1, 1, 1],
                [1, 1, 1, 1],
                [1, 1, 1, 1]]], dtype=int32)

In [62]: d = np.zeros_like(c, dtype='f16', order='C') ③
d
Out[62]: array([[[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]],

               [[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]]], dtype=float128)

In [63]: d = np.ones_like(c, dtype='f16', order='C') ③
d
Out[63]: array([[[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]],

               [[1., 1., 1., 1.],
                [1., 1., 1., 1.],
                [1., 1., 1., 1.]]], dtype=float128)

In [64]: e = np.empty((2, 3, 2)) ④
e
Out[64]: array([[[0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 0.00000000e+000]],

               [[0.00000000e+000, 0.00000000e+000],
                [0.00000000e+000, 7.49874326e-247],
                [1.28822975e-231, 4.33190018e-311]])

In [65]: f = np.empty_like(c) ④
f
Out[65]: array([[ 0,      0,      0,      0],
                [ 0,      0,      0,      0],
                [ 0,      0,      0,      0]],

```

```
[[ 0,          0,          0,          0],
 [ 0,          0, 740455269, 1936028450],
 [ 0, 268435456, 1835316017, 2041]]], dtype=int32)
```

In [66]: `np.eye(5)` ⑤

```
Out[66]: array([[1., 0., 0., 0., 0.],
 [0., 1., 0., 0., 0.],
 [0., 0., 1., 0., 0.],
 [0., 0., 0., 1., 0.],
 [0., 0., 0., 0., 1.]])
```

In [67]: `g = np.linspace(5, 15, 12)` ⑥

```
g
Out[67]: array([ 5.          ,  5.90909091,  6.81818182,  7.72727273,
 8.63636364,  9.54545455, 10.45454545, 11.36363636,
12.27272727, 13.18181818, 14.09090909, 15.          ])
```

- ① Создание массива `ndarray`, предварительно заполненного нулями.
- ② Создание массива `ndarray`, предварительно заполненного единицами.
- ③ Создание нового массива аналогичной формы с заполнением нулями либо единицами.
- ④ Создание пустого массива, не заполненного ничем (числовые значения зависят от содержимого занимаемых массивом ячеек памяти).
- ⑤ Создание единичной матрицы, заполненной единицами по диагонали.
- ⑥ Создание одномерного массива с равномерными числовыми интервалами (значения зависят от параметров `start`, `end` и `num`).

Все используемые методы поддерживают следующие параметры.

**shape**

Целое число, последовательность целочисленных значений либо ссылка на другой объект `ndarray`.

**dtype** (необязательный)

Один из поддерживаемых библиотекой NumPy типов данных для объектов `ndarray`.

**order** (необязательный)

Порядок хранения элементов массива в памяти: C — как принято в языке C (построчно), F — как принято в языке Fortran (по столбцам).



Теперь должно быть очевидно, чем именно подход к созданию массивов NumPy отличается от подхода на основе списков.

- Объект `ndarray` изначально *многомерный*.
- Объект `ndarray` *неизменяемый*, и его длина фиксирована.
- В объекте `ndarray` допускается хранить только *однотипные данные* (`np.dtype`).

Классу `array` свойственна только последняя характеристика (поддержка однотипных данных).

Назначение параметра `order` будет рассмотрено позже. Наиболее популярные значения параметра `np.dtype` (базовые типы данных, допустимые в NumPy) перечислены в табл. 4.1.

Таблица 4.1. Значения параметра `dtype` в NumPy

dtype	Описание	Пример
?	Булево значение	? (True или False)
i	Целое со знаком	i8 (64 разряда)
u	Целое без знака	u8 (64 разряда)
f	Число с плавающей точкой	f8 (64 разряда)
c	Комплексное число с плавающей точкой	c32 (256 разрядов)
m	timedelta	m (64 разряда)
M	datetime	M (64 разряда)
O	Объект	O (указатель на объект)
U	Unicode-строка	U24 (24 символа Unicode)
V	Неструктурированные данные (пустой тип)	V12 (12-байтовый блок данных)

## Метаинформация

Каждый объект `ndarray` обладает целым рядом полезных атрибутов.

```
In [68]: g.size ❶  
Out[68]: 12
```

```
In [69]: g.itemsize ❷  
Out[69]: 8
```

```
In [70]: g.ndim ❸  
Out[70]: 1
```

```
In [71]: g.shape ④
```

```
Out[71]: (12,)
```

```
In [72]: g.dtype ⑤
```

```
Out[72]: dtype('float64')
```

```
In [73]: g.nbytes ⑥
```

```
Out[73]: 96
```

- ① Количество элементов массива.
- ② Количество байтов памяти, выделяемых для хранения отдельного элемента.
- ③ Количество размерностей.
- ④ Форма массива.
- ⑤ Тип элементов.
- ⑥ Количество байтов памяти, выделенных для хранения всего массива.

## Изменение формы и размера массива

Несмотря на то что объекты `ndarray` сами по себе неизменяемые, в NumPy предусмотрено несколько возможностей изменить форму или размер массива. При *изменении формы* создается другое представление тех же самых данных, тогда как при *изменении размера* создается *новый* (временный) объект. Сначала рассмотрим несколько примеров изменения формы массива.

```
In [74]: g = np.arange(15)
```

```
In [75]: g
```

```
Out[75]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
In [76]: g.shape ①
```

```
Out[76]: (15,)
```

```
In [77]: np.shape(g) ①
```

```
Out[77]: (15,)
```

```
In [78]: g.reshape((3, 5)) ②
```

```
Out[78]: array([[ 0,  1,  2,  3,  4],
```

```
[ 5,  6,  7,  8,  9],  
[10, 11, 12, 13, 14]])
```

```
In [79]: h = g.reshape((5, 3)) ❸
```

h

```
Out[79]: array([[ 0,  1,  2],  
                [ 3,  4,  5],  
                [ 6,  7,  8],  
                [ 9, 10, 11],  
                [12, 13, 14]])
```

```
In [80]: h.T ❹
```

```
Out[80]: array([[ 0,  3,  6,  9, 12],  
                [ 1,  4,  7, 10, 13],  
                [ 2,  5,  8, 11, 14]])
```

```
In [81]: h.transpose() ❹
```

```
Out[81]: array([[ 0,  3,  6,  9, 12],  
                [ 1,  4,  7, 10, 13],  
                [ 2,  5,  8, 11, 14]])
```

- ❶ Форма исходного объекта `ndarray`.
- ❷ Преобразование в двухмерный массив.
- ❸ Создание нового объекта `ndarray`.
- ❹ Транспонирование нового массива.

В процессе изменения формы общее количество элементов массива остается прежним, а при изменении размера оно либо увеличивается, либо уменьшается. Ниже приведено несколько примеров изменения размера массива.

```
In [82]: g
```

```
Out[82]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  
                12, 13, 14])
```

```
In [83]: np.resize(g, (3, 1))❶
```

```
Out[83]: array([[0],  
                [1],  
                [2]])
```

```
In [84]: np.resize(g, (1, 5)) ❶
```

```
Out[84]: array([[0, 1, 2, 3, 4]])
```

```
In [85]: np.resize(g, (2, 5)) ❶
```

```
Out[85]: array([[0, 1, 2, 3, 4],
               [5, 6, 7, 8, 9]])
```

```
In [86]: n = np.resize(g, (5, 4)) ❷
        n
```

```
Out[86]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11],
               [12, 13, 14,  0],
               [ 1,  2,  3,  4]])
```

❶ Два измерения, уменьшение размера.

❷ Два измерения, увеличение размера.

*Склеивание* — это операция, в ходе которой два массива `ndarray` комбинируются в горизонтальном или вертикальном направлении. Учитывайте, что склеиваемые массивы должны обладать одинаковыми размерностями.

```
In [87]: h
Out[87]: array([[ 0,  1,  2],
               [ 3,  4,  5],
               [ 6,  7,  8],
               [ 9, 10, 11],
               [12, 13, 14]])
```

```
In [88]: np.hstack((h, 2 * h)) ❶
Out[88]: array([[ 0,  1,  2,  0,  2,  4],
               [ 3,  4,  5,  6,  8, 10],
               [ 6,  7,  8, 12, 14, 16],
               [ 9, 10, 11, 18, 20, 22],
               [12, 13, 14, 24, 26, 28]])
```

```
In [89]: np.vstack((h, 0.5 * h)) ❷
Out[89]: array([[ 0. ,  1. ,  2. ],
               [ 3. ,  4. ,  5. ],
               [ 6. ,  7. ,  8. ],
               [ 9. , 10. , 11. ],
               [12. , 13. , 14. ],
               [ 0. ,  0.5,  1. ],
               [ 1.5,  2. ,  2.5],
               [ 3. ,  3.5,  4. ],
               [ 4.5,  5. ,  5.5],
               [ 6. ,  6.5,  7. ]])
```

- ❶ Горизонтальное склеивание двух объектов `ndarray`.
- ❷ Вертикальное склеивание двух объектов `ndarray`.

Еще одна важная операция с объектами `ndarray` — *разуплотнение*, когда многомерный массив преобразуется в одномерный. Метод `flatten()` позволяет выбрать направление операции: по строкам (`order='C'`) или по столбцам (`order='F'`).

```
In [90]: h
Out[90]: array([[ 0,  1,  2],
               [ 3,  4,  5],
               [ 6,  7,  8],
               [ 9, 10, 11],
               [12, 13, 14]])
```

```
In [91]: h.flatten() ❶
Out[91]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
               12, 13, 14])
```

```
In [92]: h.flatten(order='C') ❶
Out[92]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
               12, 13, 14])
```

```
In [93]: h.flatten(order='F') ❷
Out[93]: array([ 0,  3,  6,  9, 12,  1,  4,  7, 10, 13,  2,  5,
               8, 11, 14])
```

```
In [94]: for i in h.flat: ❸
           print(i, end=',')
           0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
```

```
In [95]: for i in h.ravel(order='C'): ❹
           print(i, end=',')
           0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
```

```
In [96]: for i in h.ravel(order='F'): ❹
           print(i, end=',')
           0,3,6,9,12,1,4,7,10,13,2,5,8,11,14,
```

- ❶ По умолчанию разуплотнение выполняется по строкам.
- ❷ Разуплотнение по столбцам.
- ❸ Атрибут `flat` позволяет перебрать массив поэлементно (в построчном порядке).
- ❹ Разуплотнение также можно выполнить с помощью метода `ravel()`.

## Булевы массивы

Логические операции и операции сравнения с объектами `ndarray` выполняются так же, как и с массивами Python базового типа, т.е. поэлементно. Результатом такой операции станет массив булевых значений (атрибут `dtype` будет равен `bool`).

```
In [97]: h
Out[97]: array([[ 0,  1,  2],
               [ 3,  4,  5],
               [ 6,  7,  8],
               [ 9, 10, 11],
               [12, 13, 14]])
```

```
In [98]: h > 8 ❶
Out[98]: array([[False, False, False],
               [False, False, False],
               [False, False, False],
               [ True,  True,  True],
               [ True,  True,  True]])
```

```
In [99]: h <= 7 ❷
Out[99]: array([[ True,  True,  True],
               [ True,  True,  True],
               [ True,  True, False],
               [False, False, False],
               [False, False, False]])
```

```
In [100]: h == 5 ❸
Out[100]: array([[False, False, False],
                [False, False,  True],
                [False, False, False],
                [False, False, False],
                [False, False, False]])
```

```
In [101]: (h == 5).astype(int) ❹
Out[101]: array([[0, 0, 0],
                [0, 0, 1],
                [0, 0, 0],
                [0, 0, 0],
                [0, 0, 0]])
```

```
In [102]: (h > 4) & (h <= 12) ❺
Out[102]: array([[False, False, False],
                [False, False,  True],
```

```
[ True, True, True],  
[ True, True, True],  
[ True, False, False]]
```

- ❶ Значение элемента больше числа?
- ❷ Значение элемента меньше или равно числу?
- ❸ Значение элемента равно числу?
- ❹ True и False заменяются на 1 и 0.
- ❺ Значение элемента больше первого числа и меньше или равно второму числу?

Такого рода булевы массивы могут применяться для индексации и выборки данных. Обратите внимание на то, как с их помощью можно разуплотнить массива.

```
In [103]: h[h > 8] ❶  
Out[103]: array([ 9, 10, 11, 12, 13, 14])
```

```
In [104]: h[(h > 4) & (h <= 12)] ❷  
Out[104]: array([ 5, 6, 7, 8, 9, 10, 11, 12])
```

```
In [105]: h[(h < 4) | (h >= 12)] ❸  
Out[105]: array([ 0, 1, 2, 3, 12, 13, 14])
```

- ❶ Все значения, большие заданного числа.
- ❷ Все значения, большие первого числа *и* меньшие либо равные второму числу.
- ❸ Все значения, меньшие первого числа *или* большие либо равные второму числу.

В контексте булевых операций особенно удобна функция `np.where()`, позволяющая выполнять действия в зависимости от истинности или ложности заданного условия. Она возвращает новый массив, совпадающий по форме с исходным массивом.

```
In [106]: np.where(h > 7, 1, 0) ❶  
Out[106]: array([[0, 0, 0],  
                [0, 0, 0],  
                [0, 0, 1],  
                [1, 1, 1],  
                [1, 1, 1]])
```

```
In [107]: np.where(h % 2 == 0, 'чет', 'нечет') ❷
Out[107]: array([[ 'чет', 'нечет', 'чет'],
                 [ 'нечет', 'чет', 'нечет'],
                 [ 'чет', 'нечет', 'чет'],
                 [ 'нечет', 'чет', 'нечет'],
                 [ 'чет', 'нечет', 'чет']], dtype='<U4')
```

```
In [108]: np.where(h <= 7, h * 2, h / 2) ❸
Out[108]: array([[ 0. ,  2. ,  4. ],
                 [ 6. ,  8. , 10. ],
                 [12. , 14. ,  4. ],
                 [ 4.5,  5. ,  5.5],
                 [ 6. ,  6.5,  7. ]])
```

- ❶ Если условие истинно, то элемент нового массива равен 1, иначе — 0.
- ❷ Если условие истинно, то элемент нового массива равен 'чет', иначе — 'нечет'.
- ❸ Если условие истинно, то элемент массива умножается на 2, иначе — делится на 2.

Дополнительные примеры выполнения операций с массивами `ndarray` будут приведены в следующих главах.

## Скорость выполнения операций

Вскоре мы перейдем к рассмотрению структурированных массивов `NumPy`, но перед этим попробуем разобраться, насколько эффективно применение специализированных объектов с точки зрения производительности.

В качестве простого примера возьмем операцию генерирования массива размером  $5000 \times 5000$ , который заполняется псевдослучайными числами с нормальным распределением, после чего вычисляется их сумма. Сначала рассмотрим решение на основе списковых включений, т.е. стандартными средствами Python.

```
In [109]: import random
I = 5000
```

```
In [110]: %time mat = [[random.gauss(0, 1) for j in range(I)] \
                      for i in range(I)] ❶
CPU times: user 17.1 s, sys: 361 ms, total: 17.4 s
Wall time: 17.4 s
```



```
In [111]: mat[0][:5] ❷
Out[111]: [-0.40594967782329183,
           -1.357757478015285,
            0.05129566894355976,
           -0.8958429976582192,
            0.6234174778878331]
```

```
In [112]: %time sum([sum(l) for l in mat]) ❸
CPU times: user 142 ms, sys: 1.69 ms, total: 144 ms
Wall time: 143 ms
Out[112]: -3561.944965714259
```

```
In [113]: import sys
           sum([sys.getsizeof(l) for l in mat]) ❹
Out[113]: 215200000
```

- ❶ Создание матрицы на основе спискового включения.
- ❷ Выборка случайных чисел.
- ❸ Сначала вычисляются суммы отдельных списков, а затем подсчитывается сумма полученных сумм.
- ❹ Суммирование объемов памяти, выделенных под все объекты `list`.

Теперь рассмотрим, как аналогичная задача решается средствами библиотеки NumPy. Для удобства воспользуемся его подпакетом `random`, включающим функции для инициализации объекта `ndarray` и его заполнения псевдослучайными числами.

```
In [114]: %time mat = np.random.standard_normal((I, I)) ❶
CPU times: user 1.01 s, sys: 200 ms, total: 1.21 s
Wall time: 1.21 s

In [115]: %time mat.sum() ❷
CPU times: user 29.7 ms, sys: 1.15 ms, total: 30.8 ms
Wall time: 29.4 ms
Out[115]: -186.12767026606448

In [116]: mat.nbytes ❸
Out[116]: 200000000

In [117]: sys.getsizeof(mat) ❸
Out[117]: 200000112
```

- ❶ Создание массива случайных чисел с нормальным распределением; выполняется приблизительно в 14 раз быстрее, чем в случае обычного массива.
- ❷ Вычисление суммы всех значений объекта `ndarray`; выполняется приблизительно в 4,5 раза быстрее, чем в случае обычного массива.
- ❸ Благодаря пакету NumPy также обеспечивается определенная экономия памяти, поскольку объем памяти, выделяемой для хранения объекта `ndarray`, намного меньше объема самих данных.



### Преимущества массивов NumPy

Применение библиотеки NumPy для работы с массивами позволяет получить более компактный и понятный код, который выполняется намного быстрее в сравнении с чистым кодом Python.

## Структурированные массивы NumPy

Специализированные инструменты, реализованные в классе `ndarray`, дают существенные преимущества при работе с массивами. Однако чрезмерная специализированность становится помехой для многих приложений и алгоритмов. Поэтому в библиотеке NumPy поддерживаются также *структурированные объекты ndarray* и *объекты записей гесагау*, в которых можно указывать разные значения `dtype` для каждого столбца. Рассмотрим следующий пример инициализации структурированного объекта `ndarray`.

```
In [118]: dt = np.dtype([('Name', 'S10'), ('Age', 'i4'),  
                        ('Height', 'f'), ('Children/Pets', 'i4', 2)]) ❶
```

```
In [119]: dt ❶  
Out[119]: dtype([('Name', 'S10'), ('Age', '<i4'), ('Height', '<f4'),  
                ('Children/Pets', '<i4', (2,))])
```

```
In [120]: dt = np.dtype({'names': ['Name', 'Age', 'Height',  
                                   'Children/Pets'], 'formats':  
                                   'O int float int,int'.split()}) ❷
```

```
In [121]: dt ❷  
Out[121]: dtype([('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'),  
                ('Children/Pets', [('f0', '<i8'), ('f1', '<i8')])])
```

```
In [122]: s = np.array([('Smith', 45, 1.83, (0, 1)),  
                        ('Jones', 53, 1.72, (2, 2))], dtype=dt) ❸
```

```
In [123]: s ❸  
Out[123]: array([('Smith', 45, 1.83, (0, 1)), ('Jones', 53, 1.72,  
        (2, 2))], dtype=[('Name', 'O'), ('Age', '<i8'),  
        ('Height', '<f8'), ('Children/Pets', [('f0', '<i8'),  
        ('f1', '<i8')])])
```

```
In [124]: type(s) ❹  
Out[124]: numpy.ndarray
```

- ❶ Составной атрибут `dtype`.
- ❷ Альтернативный синтаксис для тех же самых целей.
- ❸ Создание структурированного объекта `ndarray`, включающего две записи.
- ❹ Объект по-прежнему имеет тип `ndarray`.

Все это напоминает процесс инициализации таблиц в реляционной базе данных, когда задаются имена и типы столбцов, а также дополнительная информация (например, максимальное количество символов, которое может содержать строковый объект). Теперь к столбцам можно обращаться по названиям, а к строкам — по индексам.

```
In [125]: s['Name'] ❶  
Out[125]: array(['Smith', 'Jones'], dtype=object)
```

```
In [126]: s['Height'].mean() ❷  
Out[126]: 1.775
```

```
In [127]: s[0] ❸  
Out[127]: ('Smith', 45, 1.83, (0, 1))
```

```
In [128]: s[1]['Age'] ❹  
Out[128]: 53
```

- ❶ Выбор столбца по названию.
- ❷ Вызов метода для выбранного столбца.
- ❸ Выбор отдельной записи.
- ❹ Выбор поля в записи.

Подводя итог, можно сказать следующее: структурированные массивы представляют собой обобщение объектов `ndarray` в том смысле, что уникальность типов данных в них соблюдается только *в пределах столбца*, как в таблицах реляционных баз данных. Основным преимуществом таких массивов становится то, что теперь элементом столбца не обязательно должно быть значение одного из поддерживаемых в NumPy типов — это может быть и другой многомерный объект.



### Структурированные массивы

Помимо обычных массивов в NumPy поддерживаются также структурированные массивы (и массивы записей), с помощью которых можно формировать табличные структуры с именованными столбцами различного типа. Это позволяет сочетать преимущества объектов `ndarray` (синтаксис, методы, производительность) с управлением реляционными таблицами.

## Векторизация кода

*Векторизация* — это стратегия получения более компактного и потенциально быстрее эффективного кода. Идея заключается в том, что операция или функция применяется сразу ко всему структурированному объекту, без циклической обработки каждого его элемента. В Python определенные возможности векторизации реализуются средствами функционального программирования, в частности функциями `map()` и `filter()`. Но в библиотеке NumPy векторизация поддерживается повсеместно.

### Основные способы векторизации

Как было показано в предыдущем разделе, простые математические операции с массивами `ndarray` (например, вычисление суммы всех элементов) можно выполнять непосредственно над самим объектом с помощью встроенных методов или универсальных функций. Но поддерживаются и более общие векторизованные операции. Ниже продемонстрировано, как выполнить поэлементное сложение двух массивов `ndarray`.

```
In [129]: np.random.seed(100)
          r = np.arange(12).reshape((4, 3)) ❶
          s = np.arange(12).reshape((4, 3)) * 0.5 ❷
```

```
In [130]: r ❶
```

```
Out[130]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])
```

```
In [131]: s ❷
```

```
Out[131]: array([[0. , 0.5, 1. ],
                 [1.5, 2. , 2.5],
                 [3. , 3.5, 4. ],
                 [4.5, 5. , 5.5]])
```

```
In [132]: r + s ❸
```

```
Out[132]: array([[ 0. ,  1.5,  3. ],
                 [ 4.5,  6. ,  7.5],
                 [ 9. , 10.5, 12. ],
                 [13.5, 15. , 16.5]])
```

- ❶ Первый массив `ndarray`, заполненный случайными числами.
- ❷ Второй массив `ndarray`, заполненный теми же случайными числами с коэффициентом 0.5.
- ❸ Поэлементное сложение массивов в виде векторизованной операции (цикл не потребовался).

В NumPy также поддерживается *трансляция* — операция, выполняемая над объектами разной формы. Рассмотрим пример.

```
In [133]: r + 3 ❶
```

```
Out[133]: array([[ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11],
                 [12, 13, 14]])
```

```
In [134]: 2 * r ❷
```

```
Out[134]: array([[ 0,  2,  4],
                 [ 6,  8, 10],
                 [12, 14, 16],
                 [18, 20, 22]])
```

```
In [135]: 2 * r + 3 ❸
```

```
Out[135]: array([[ 3,  5,  7],
                 [ 9, 11, 13],
                 [15, 17, 19],
                 [21, 23, 25]])
```

- ❶ При выполнении операции скалярного сложения скаляр транслируется в массив и добавляется к каждому элементу искомого массива.
- ❷ При выполнении операции скалярного умножения скаляр также транслируется в массив и умножается на каждый элемент искомого массива.
- ❸ В данном линейном преобразовании объединяются обе операции.

В подобных операциях могут участвовать объекты `ndarray` разной формы (но только при наличии совпадающей размерности).

```
In [136]: r
Out[136]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])
```

```
In [137]: r.shape
Out[137]: (4, 3)
```

```
In [138]: s = np.arange(0, 12, 4) ❶
          s ❶
Out[138]: array([0, 4, 8])
```

```
In [139]: r + s ❷
Out[139]: array([[ 0,  5, 10],
                 [ 3,  8, 13],
                 [ 6, 11, 16],
                 [ 9, 14, 19]])
```

```
In [140]: s = np.arange(0, 12, 3) ❸
          s ❸
Out[140]: array([0, 3, 6, 9])
```

```
In [141]: r + s ❹
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-141-1890b26ec965> in <module>()
----> 1 r + s
```

```
ValueError: operands could not be broadcast together
          with shapes (4,3) (4,)
```

```
In [142]: r.transpose() + s ❺
```

```
Out[142]: array([[ 0,  6, 12, 18],
                 [ 1,  7, 13, 19],
                 [ 2,  8, 14, 20]])
```

```
In [143]: sr = s.reshape(-1, 1) ⑥
          sr
```

```
Out[143]: array([[0],
                 [3],
                 [6],
                 [9]])
```

```
In [144]: sr.shape ⑥
```

```
Out[144]: (4, 1)
```

```
In [145]: r + s.reshape(-1, 1) ⑥
```

```
Out[145]: array([[ 0,  1,  2],
                 [ 6,  7,  8],
                 [12, 13, 14],
                 [18, 19, 20]])
```

- ❶ Новый одномерный массив `ndarray` длиной 3.
- ❷ Объекты `r` (матрица) и `s` (вектор) суммируются напрямую.
- ❸ Еще один одномерный массив `ndarray` длиной 4.
- ❹ Длина нового вектора `s` теперь отличается от второй размерности объекта `r`.
- ❺ Транспонирование объекта `r` снова делает возможной векторизованную операцию сложения.
- ❻ Альтернативный вариант — поменять форму объекта `s` на `(4, 1)` (результатирующие матрицы получаются разными).

К объектам `ndarray` можно применять пользовательские функции Python. В подобного рода операциях массивы могут использоваться так же, как значения типа `int` или `float`. Рассмотрим следующую функцию.

```
In [146]: def f(x):
          return 3 * x + 5 ❶
```

```
In [147]: f(0.5) ❷
```

```
Out[147]: 6.5
```

```
In [148]: f(r) ❸
```

```
Out[148]: array([[ 5,  8, 11],
                 [14, 17, 20],
                 [23, 26, 29],
                 [32, 35, 38]])
```

- ❶ Простая функция, выполняющая линейное преобразование параметра `x`.
- ❷ Функция `f()` применяется к значению типа `float`.
- ❸ Та же самая функция применяется к массиву `ndaarray` векторизованно и поэлементно.

Как видите, функция `f()` применяется к объекту NumPy поэлементно. В этом смысле программист не избегает циклов. Они просто не создаются на уровне кода Python, но применяются на уровне библиотеки NumPy, где соответствующий код оптимизирован на языке C, а потому работает быстрее, чем чистый код Python. Именно этим объясняется высокая производительность операций с массивами в NumPy.

## Эффективное использование памяти

Рассмотренный ранее метод `np.zeros()`, с помощью которого инициализируется объект `ndaarray`, поддерживает необязательный аргумент, задающий способ хранения массива в памяти. Грубо говоря, этот аргумент определяет, какие элементы оказываются смежными в памяти. В случае небольших массивов разницы с точки зрения производительности не будет никакой. Но в определенных финансовых приложениях, работающих с большими массивами, разница начинает проявляться (см., например, статью *Memory layout of multi-dimensional arrays*, доступную по адресу <https://bit.ly/2K8rujN>).

Чтобы понять, насколько сильно способ хранения массива в памяти влияет на производительность финансовых вычислений, рассмотрим следующий код.

```
In [149]: x = np.random.standard_normal((1000000, 5)) ❶
```

```
In [150]: y = 2 * x + 3 ❷
```

```
In [151]: C = np.array((x, y), order='C') ❸
```

```
In [152]: F = np.array((x, y), order='F') ❹
```

```
In [153]: x = 0.0; y = 0.0 ❺
```



```
In [154]: C[:2].round(2) ❹
Out[154]: array([[[-1.75,  0.34,  1.15, -0.25,  0.98],
                  [ 0.51,  0.22, -1.07, -0.19,  0.26],
                  [-0.46,  0.44, -0.58,  0.82,  0.67],
                  ...,
                  [-0.05,  0.14,  0.17,  0.33,  1.39],
                  [ 1.02,  0.3 , -1.23, -0.68, -0.87],
                  [ 0.83, -0.73,  1.03,  0.34, -0.46]]],

                [[[-0.5 ,  3.69,  5.31,  2.5 ,  4.96],
                  [ 4.03,  3.44,  0.86,  2.62,  3.51],
                  [ 2.08,  3.87,  1.83,  4.63,  4.35],
                  ...,
                  [ 2.9 ,  3.28,  3.33,  3.67,  5.78],
                  [ 5.04,  3.6 ,  0.54,  1.65,  1.26],
                  [ 4.67,  1.54,  5.06,  3.69,  2.07]]]])
```

- ❶ Двухмерный объект `ndarray` с резкой асимметрией размерностей.
- ❷ Линейное преобразование исходного массива.
- ❸ Создание двухмерного объекта `ndarray` с порядком заполнения элементов 'C' (построчно).
- ❹ Создание двухмерного объекта `ndarray` с порядком заполнения элементов 'F' (по столбцам).
- ❺ Очистка памяти.
- ❻ Фрагмент массива `C`.

Теперь попробуем поработать с обоими массивами и оценим скорость выполнения типичных операций.

```
In [155]: %timeit C.sum() ❶
4.36 ms ± 89.3 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

```
In [156]: %timeit F.sum() ❶
4.21 ms ± 71.4 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

```
In [157]: %timeit C.sum(axis=0) ❷
17.9 ms ± 776 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

```
In [158]: %timeit C.sum(axis=1) ③
          35.1 ms ± 999 µs per loop (mean ± std. dev. of 7 runs,
          10 loops each)

In [159]: %timeit F.sum(axis=0) ②
          83.8 ms ± 2.63 ms per loop (mean ± std. dev. of 7 runs,
          10 loops each)

In [160]: %timeit F.sum(axis=1) ③
          67.9 ms ± 5.16 ms per loop (mean ± std. dev. of 7 runs,
          10 loops each)

In [161]: F = 0.0; C = 0.0
```

- ① Вычисление суммы всех элементов массива.
- ② Вычисление суммы элементов по строкам (большое количество).
- ③ Вычисление суммы элементов по столбцам (малое количество).

Полученные результаты позволяют сделать следующие выводы.

- При суммировании *всех элементов* массива способ его хранения в памяти не играет особой роли.
- Для массивов с порядком заполнения 'C' суммирование выполняется быстрее как по строкам, так и по столбцам (абсолютное преимущество в скорости).
- Для массивов с порядком заполнения 'C' (построчное хранение) суммирование выполняется быстрее по строкам, чем по столбцам.
- Для массивов с порядком заполнения 'F' (хранение по столбцам) суммирование выполняется быстрее по столбцам, чем по строкам.

## Резюме

NumPy — это основной пакет для работы с массивами в Python. Предоставляемый им класс `ndarray` специально предназначен для эффективных матричных вычислений. Наличие удобных методов и универсальных функций, поддерживающих векторизованные операции, позволяет избежать медленных циклов на уровне кода Python. Все это пригодится нам при работе с пакетом `pandas` и его классом `DataFrame` в следующей главе.

## Дополнительные ресурсы

Множество полезной информации о возможностях пакета NumPy можно найти на официальном сайте ([www.numpy.org](http://www.numpy.org)).

Хорошим подспорьем в изучении NumPy будут следующие книги.

- McKinney, Wes. *Python for Data Analysis* (2017, O'Reilly).
- VanderPlas, Jake. *Python Data Science Handbook* (2016, O'Reilly).

---

# Анализ данных с помощью библиотеки pandas

Факты! Факты! Факты! Я не могу лепить кирпичи без глины!

*Шерлок Холмс*

Эта глава посвящена библиотеке `pandas`, предлагающей инструменты анализа табличных данных. Библиотека не только содержит полезные классы и функции, но и служит оболочкой к другим пакетам, что делает ее незаменимой для анализа финансовых данных.

В этой главе рассматриваются следующие фундаментальные структуры данных.

Тип объекта	Описание	Применение
<code>DataFrame</code>	Двухмерный индексированный объект данных	Табличные данные с разбивкой по столбцам
<code>Series</code>	Одномерный индексированный объект данных	Временные ряды

---

Общая структура главы такова.

## *Класс `DataFrame`*

В этом разделе описываются основные возможности класса `DataFrame` на примере небольших наборов данных. Также показано, как преобразовать объект `ndarray` библиотеки `NumPy` в объект `DataFrame`.

## *Основные аналитические возможности и инструменты визуализации*

Мы познакомимся только с основными способами анализа и визуализации данных (более подробно эти темы рассматриваются в следующих главах).

## *Класс `Series`*

Данный раздел содержит краткое описание класса `Series`, который представляет собой частный случай класса `DataFrame`, т.к. предназначен для работы с одномерными рядами.

### *Группирование данных*

Одна из ключевых особенностей класса `DataFrame` — возможность группировки данных по одному или нескольким столбцам.

### *Сложные операции извлечения данных*

В этом разделе показано, как применять сложные условия для извлечения данных из объекта `DataFrame`.

### *Конкатенация, соединение и слияние данных*

В процессе анализа часто приходится комбинировать различные наборы данных. Библиотека `pandas` предлагает множество способов решения этой задачи.

### *Производительность вычислений*

Следуя идеологии Python, библиотека `pandas` позволяет решать одни и те же задачи несколькими способами. В этом разделе будет показано, как выбрать наиболее эффективный вариант.

## Класс `DataFrame`

Основа библиотеки `pandas` — класс `DataFrame`, предназначенный для эффективной обработки табличных данных (т.е. данных, разбитых по столбцам). В частности, в нем имеются средства именования столбцов, а также поддерживается гибкая индексация записей, как в таблицах реляционных баз данных или электронных таблицах Excel.

В этом разделе вы познакомитесь с основными возможностями класса `DataFrame`. Функциональность класса настолько широка, что охватить ее в одной главе не получится. Мы будем неоднократно возвращаться к нему в последующих главах.

## Знакомство с классом `DataFrame`

Класс `DataFrame` изначально предназначен для работы с индексированными и размеченными данными, напоминающими таблицы реляционных баз данных или рабочие листы Excel. Для начала рассмотрим, как создается объект `DataFrame`.

```
In [1]: import pandas as pd ❶
```

```
In [2]: df = pd.DataFrame([10, 20, 30, 40], ❷  
                           columns=['Целые'], ❸  
                           index=['a', 'b', 'c', 'd']) ❹
```

```
In [3]: df ⑤
```

```
Out[3]:
```

	Целые
a	10
b	20
c	30
d	40

- ❶ Импорт пакета `pandas`.
- ❷ Данные представляют собой список.
- ❸ Название столбца.
- ❹ Индексы (метки).
- ❺ Вывод содержимого объекта `DataFrame`, включая название столбца и метки.

В этом простом примере продемонстрированы основные возможности хранения данных в объектах `DataFrame`.

- Сами данные могут быть какого угодно типа (`list`, `tuple`, `ndarray`, `dict` и др.).
- Данные группируются в столбцы, которым присваиваются имена (подписи).
- Данные снабжаются индексами, которые могут иметь разный формат (числа, строки, дата/время и т.п.).

Работать с объектом `DataFrame` удобнее и эффективнее, чем с обычным объектом `ndarray`, который накладывает немало ограничений, когда нужно, к примеру, увеличить существующий массив. С точки зрения производительности вычислений возможности этих двух классов примерно сопоставимы. Ниже приведено несколько примеров выполнения типичных операций с объектом `DataFrame`.

```
In [4]: df.index ❶
```

```
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [5]: df.columns ❷
```

```
Out[5]: Index(['Целые'], dtype='object')
```

```
In [6]: df.loc['c'] ❸
```

```
Out[6]: Целые      30  
       Name: c, dtype: int64
```

```
In [7]: df.loc[['a', 'd']] ❷
```

```
Out[7]:
```

	Целые
a	10
d	40

```
In [8]: df.iloc[1:3] ❸
```

```
Out[8]:
```

	Целые
b	20
c	30

```
In [9]: df.sum() ❹
```

```
Out[9]: Целые      100  
       dtype: int64
```

```
In [10]: df.apply(lambda x: x ** 2) ❺
```

```
Out[10]:
```

	Целые
a	100
b	400
c	900
d	1600

```
In [11]: df ** 2 ❻
```

```
Out[11]:
```

	Целые
a	100
b	400
c	900
d	1600

- ❶ Атрибут `index` и объект `Index`.
- ❷ Атрибут `columns` и объект `Index`.
- ❸ Выбор значения, соответствующего индексу `c`.
- ❹ Выбор двух значений, соответствующих индексам `a` и `d`.
- ❺ Выбор второй и третьей строки по номерам индексов.
- ❻ Вычисление суммы значений столбца.

- ⑦ Применение метода `apply()` для вычисления квадратов значений векторизованным способом.
- ⑧ Выполнение векторизованной операции непосредственно над объектом `DataFrame`.

В отличие от массивов `ndarray` объект `DataFrame` можно увеличивать по любым измерениям.

```
In [12]: df['Вещественные'] = (1.5, 2.5, 3.5, 4.5) ❶
```

```
In [13]: df
```

```
Out[13]:
```

	Целые	Вещественные
<b>a</b>	10	1.5
<b>b</b>	20	2.5
<b>c</b>	30	3.5
<b>d</b>	40	4.5

```
In [14]: df['Вещественные'] ❷
```

```
Out[14]: a    1.5
```

```
       b    2.5
```

```
       c    3.5
```

```
       d    4.5
```

```
       Name: Вещественные, dtype: float64
```

- ❶ Добавление нового столбца с вещественными числами, представленными в виде кортежа.
- ❷ Вывод содержимого столбца вместе с индексами.

В качестве источника данных для столбца может использоваться другой объект `DataFrame`. В подобных случаях индексы согласуются автоматически.

```
In [15]: df['Имена'] = pd.DataFrame(['Ив', 'Сандра', 'Лилли',  
                                     'Генри'], index=['d', 'a', 'b', 'c']) ❶
```

```
In [16]: df
```

```
Out[16]:
```

	Целые	Вещественные	Имена
<b>a</b>	10	1.5	Сандра
<b>b</b>	20	2.5	Лилли
<b>c</b>	30	3.5	Генри
<b>d</b>	40	4.5	Ив

- ❶ Создание нового столбца на основе другого объекта `DataFrame`.



Схожим образом работает операция присоединения данных. Однако следует опасаться неприятного побочного эффекта — замены индексов числовым диапазоном.

```
In [17]: df.append({'Целые': 100, 'Вещественные': 5.75,  
                  'Имена': 'Джил'}, ignore_index=True) ❶
```

```
Out[17]:
```

	Целые	Вещественные	Имена
0	10	1.50	Сандра
1	20	2.50	Лилли
2	30	3.50	Генри
3	40	4.50	Ив
4	100	5.75	Джил

```
In [18]: df = df.append(pd.DataFrame({'Целые': 100, 'Вещественные':  
                                     5.75, 'Имена': 'Джил'},  
                                index=['y',])) ❷
```

```
In [19]: df
```

```
Out[19]:
```

	Целые	Вещественные	Имена
a	10	1.50	Сандра
b	20	2.50	Лилли
c	30	3.50	Генри
d	40	4.50	Ив
y	100	5.75	Джил

```
In [20]: df = df.append(pd.DataFrame({'Имена': 'Лиз'}, index=['z',]),  
                        sort=False) ❸
```

```
In [21]: df
```

```
Out[21]:
```

	Целые	Вещественные	Имена
a	10.0	1.50	Сандра
b	20.0	2.50	Лилли
c	30.0	3.50	Генри
d	40.0	4.50	Ив
y	100.0	5.75	Джил
z	NaN	NaN	Лиз

```
In [22]: df.dtypes ❹
```

```
Out[22]: Целые          float64  
Вещественные    float64  
Имена          object  
dtype: object
```

- ❶ Добавление новой записи в виде словаря, при этом исходные индексы безвозвратно теряются.
- ❷ Добавление новой записи на основе объекта `DataFrame`, включающего индексную информацию. Исходные индексы остаются неизменными.
- ❸ Добавление неполной записи приводит к появлению значений `NaN`.
- ❹ Просмотр типов столбцов. Это напоминает работу со структурированным объектом `ndarray`.

Несмотря на появление пустых значений, методы объекта `DataFrame` по-прежнему работают.

```
In [23]: df[['Целые', 'Вещественные']].mean() ❶
Out[23]: Целые          40.00
         Вещественные   3.55
         dtype: float64
```

```
In [24]: df[['Целые', 'Вещественные']].std() ❷
Out[24]: Целые          35.355339
         Вещественные   1.662077
         dtype: float64
```

- ❶ Вычисление среднего для двух столбцов (строки со значениями `NaN` игнорируются).
- ❷ Вычисление среднеквадратического отклонения для двух столбцов (строки со значениями `NaN` игнорируются).

## Расширенные возможности класса `DataFrame`

Ниже рассматривается пример, в котором используется объект `ndarray`, содержащий массив случайных чисел с нормальным распределением. Это позволит нам исследовать дополнительные возможности библиотеки `pandas`, в частности функцию `DatetimeIndex()`, предназначенную для обработки временных рядов.

```
In [25]: import numpy as np
```

```
In [26]: np.random.seed(100)
```

```
In [27]: a = np.random.standard_normal((9, 4))
```

```
In [28]: a
```

```
Out[28]: array([[ -1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
 [  0.98132079,  0.51421884,  0.22117967, -1.07004333],
 [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
 [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
 [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
 [  1.61898166,  1.54160517, -0.25187914, -0.84243574],
 [  0.18451869,  0.9370822 ,  0.73100034, -1.36155613],
 [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
 [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

Объект `DataFrame` можно создавать по-разному, в том числе на основе объекта `ndarray`. Такое решение удобно, поскольку структура исходного массива не меняется, к нему лишь добавляется метainформация (индексы). Это типичный способ импорта числовых значений в финансовых и научных приложениях.

```
In [29]: df = pd.DataFrame(a) ❶
```

```
In [30]: df
```

```
Out[30]:
```

	0	1	2	3
0	-1.749765	0.342680	1.153036	-0.252436
1	0.981321	0.514219	0.221180	-1.070043
2	-0.189496	0.255001	-0.458027	0.435163
3	-0.583595	0.816847	0.672721	-0.104411
4	-0.531280	1.029733	-0.438136	-1.118318
5	1.618982	1.541605	-0.251879	-0.842436
6	0.184519	0.937082	0.731000	1.361556
7	-0.326238	0.055676	0.222400	-1.443217
8	-0.756352	0.816454	0.750445	-0.455947

❶ Создание объекта `DataFrame` на основе объекта `ndarray`.

В табл. 5.1 перечислены параметры, передаваемые функции `DataFrame()`.

*Таблица 5.1. Параметры функции `DataFrame()`*

Параметр	Формат	Описание
<code>data</code>	<code>ndarray</code> / <code>dict</code> / <code>DataFrame</code>	Данные для объекта <code>DataFrame</code> . Объект <code>dict</code> может включать объекты <code>Series</code> , <code>ndarray</code> и <code>list</code>
<code>index</code>	<code>Index</code> /массив	Список индексов; по умолчанию — <code>range(n)</code>
<code>columns</code>	<code>Index</code> /массив	Список имен столбцов; по умолчанию — <code>range(n)</code>
<code>dtype</code>	<code>dtype</code> , по умолчанию <code>None</code>	Тип данных столбцов; по умолчанию подбирается автоматически
<code>copy</code>	<code>bool</code> , по умолчанию <code>None</code>	Флаг копирования данных из источника

Термином “массив” здесь обозначается структура, подобная объекту `ndarray`, например список. `Index` — это экземпляр класса `Index` библиотеки `pandas`.

Как было показано выше, имена столбцов объекта `DataFrame` можно непосредственно задать в виде списка, содержащего нужное число элементов. Другими словами, задавать или менять атрибуты объекта `DataFrame` совсем не сложно.

```
In [31]: df.columns = ['№1', '№2', '№3', '№4'] ❶
```

```
In [32]: df
```

```
Out[32]:
```

	№1	№2	№3	№4
0	-1.749765	0.342680	1.153036	-0.252436
1	0.981321	0.514219	0.221180	-1.070043
2	-0.189496	0.255001	-0.458027	0.435163
3	-0.583595	0.816847	0.672721	-0.104411
4	-0.531280	1.029733	-0.438136	-1.118318
5	1.618982	1.541605	-0.251879	-0.842436
6	0.184519	0.937082	0.731000	1.361556
7	-0.326238	0.055676	0.222400	-1.443217
8	-0.756352	0.816454	0.750445	-0.455947

```
In [33]: df['№2'].mean() ❷
```

```
Out[33]: 0.7010330941456459
```

❶ Задание имен столбцов в виде списка.

❷ Теперь столбец можно легко выбрать по имени.

Для эффективной работы с временными рядами необходимо иметь возможность использовать значения даты/времени в качестве индексов. В библиотеке `pandas` эта задача решается очень легко. Предположим, что в нашей таблице, состоящей из девяти строк и четырех столбцов, записи соответствуют датам конца месяца, начиная с января 2019 года. Соответствующий объект `DatetimeIndex` генерируется с помощью метода `date_range()`.

```
In [34]: dates = pd.date_range('2019-1-1', periods=9, freq='M') ❶
```

```
In [35]: dates
```

```
Out[35]: DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31',  
                        '2019-04-30', '2019-05-31', '2019-06-30',  
                        '2019-07-31', '2019-08-31', '2019-09-30'],  
                        dtype='datetime64[ns]', freq='M')
```

❶ Создание объекта `DatetimeIndex`.

Параметры метода `date_range()` описаны в табл. 5.2.

Таблица 5.2. Параметры метода `date_range()`

Параметр	Формат	Описание
<code>start</code>	<code>string/datetime</code>	Начальная дата диапазона
<code>end</code>	<code>string/datetime</code>	Конечная дата диапазона
<code>periods</code>	<code>integer/None</code>	Количество периодов (если параметр <code>start</code> или <code>end</code> равен <code>None</code> )
<code>freq</code>	<code>string/DateOffset</code>	Строка, задающая длину периода (например, <code>5D</code> означает 5 дней)
<code>tz</code>	<code>string/None</code>	Название часового пояса в случае локализованного индекса
<code>normalize</code>	<code>bool</code> , по умолчанию <code>None</code>	Флаг нормализации дат <code>start</code> и <code>end</code> , чтобы отсчет шел с полуночи
<code>name</code>	<code>string</code> , по умолчанию <code>None</code>	Название создаваемого индекса

В следующем примере только что созданный объект `DatetimeIndex` применяется к имеющемуся массиву.

```
In [36]: df.index = dates
```

```
In [37]: df
```

```
Out[37]:
```

	№1	№2	№3	№4
<b>2019-01-31</b>	-1.749765	0.342680	1.153036	-0.252436
<b>2019-02-28</b>	0.981321	0.514219	0.221180	-1.070043
<b>2019-03-31</b>	-0.189496	0.255001	-0.458027	0.435163
<b>2019-04-30</b>	-0.583595	0.816847	0.672721	-0.104411
<b>2019-05-31</b>	-0.531280	1.029733	-0.438136	-1.118318
<b>2019-06-30</b>	1.618982	1.541605	-0.251879	-0.842436
<b>2019-07-31</b>	0.184519	0.937082	0.731000	1.361556
<b>2019-08-31</b>	-0.326238	0.055676	0.222400	-1.443217
<b>2019-09-30</b>	-0.756352	0.816454	0.750445	-0.455947

При создании объектов `DatetimeIndex` с помощью метода `date_range()` можно использовать различные значения аргумента `freq` (табл. 5.3).

Таблица 5.3. Значения параметра `freq` метода `date_range()`

Единица измерения	Описание
B	Банковский день
C	Назначаемый банковский день (экспериментальная возможность)

Единица измерения	Описание
D	Календарный день
W	Неделя
M	Конец месяца
BM	Конец финансового месяца
MS	Начало месяца
BMS	Начало финансового месяца
Q	Конец квартала
BQ	Конец финансового квартала
QS	Начало квартала
BQS	Начало финансового квартала
A	Конец года
BA	Конец финансового года
AS	Начало года
BAS	Начало финансового года
H	Час
T	Минута
S	Секунда
L	Миллисекунда
U	Микросекунда

В некоторых случаях может потребоваться получить доступ к исходным данным массива `ndarray`. Для этого предусмотрен атрибут `values`.

```
In [38]: df.values
Out[38]: array([[ -1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
 [  0.98132079,  0.51421884,  0.22117967, -1.07004333],
 [ -0.18949583,  0.25500144, -0.45802699,  0.43516349],
 [ -0.58359505,  0.81684707,  0.67272081, -0.10441114],
 [ -0.53128038,  1.02973269, -0.43813562, -1.11831825],
 [  1.61898166,  1.54160517, -0.25187914, -0.84243574],
 [  0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
 [ -0.32623806,  0.05567601,  0.22239961, -1.443217 ],
 [ -0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

```
In [39]: np.array(df)
```

```
Out[39]: array([[ -1.74976547,  0.34268004,  1.1530358 , -0.25243604],
 [  0.98132079,  0.51421884,  0.22117967, -1.07004333],
 [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
 [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
 [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
 [  1.61898166,  1.54160517, -0.25187914, -0.84243574],
 [  0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
 [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
 [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```



### Объекты ndarray и DataFrame

Можно не только создать объект DataFrame из объекта ndarray, но и выполнить обратную операцию — создать объект ndarray из объекта DataFrame. Для этого воспользуйтесь атрибутом `values` класса DataFrame или методом `np.array()` библиотеки NumPy.

## Основные аналитические возможности

Подобно классу ndarray в NumPy, класс DataFrame библиотеки pandas содержит множество удобных методов. Для начала рассмотрим методы `info()` и `describe()`.

```
In [40]: df.info() ❶
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 9 entries, 2019-01-31 to 2019-09-30
Freq: M
Data columns (total 4 columns):
#1    9 non-null float64
#2    9 non-null float64
#3    9 non-null float64
#4    9 non-null float64
dtypes: float64(4)
memory usage: 360.0 bytes
```

```
In [41]: df.describe() ❷
Out[41]:
```

	#1	#2	#3	#4
<b>count</b>	9.000000	9.000000	9.000000	9.000000
<b>mean</b>	-0.150212	0.701033	0.289193	-0.387788
<b>std</b>	0.988306	0.457685	0.579920	0.877532
<b>min</b>	-1.749765	0.055676	-0.458027	-1.443217
<b>25%</b>	-0.583595	0.342680	-0.251879	-1.070043

<b>50%</b>	-0.326238	0.816454	0.222400	-0.455947
<b>75%</b>	0.184519	0.937082	0.731000	-0.104411
<b>max</b>	1.618982	1.541605	1.153036	1.361556

- ❶ Выводит метаинформацию о структуре данных, столбцах и индексах.
- ❷ Выводит статистическую сводку по числовым столбцам.

Кроме того, можно легко вычислить сумму, среднее и накопительную сумму как по столбцам, так и по строкам.

In [42]: df.sum() ❶

```
Out[42]: №1    -1.351906
        №2     6.309298
        №3     2.602739
        №4    -3.490089
        dtype: float64
```

In [43]: df.mean() ❷

```
Out[43]: №1    -0.150212
        №2     0.701033
        №3     0.289193
        №4    -0.387788
        dtype: float64
```

In [44]: df.mean(axis=0) ❷

```
Out[44]: №1    -0.150212
        №2     0.701033
        №3     0.289193
        №4    -0.387788
        dtype: float64
```

In [45]: df.mean(axis=1) ❸

```
Out[45]: 2019-01-31    -0.126621
        2019-02-28     0.161669
        2019-03-31     0.010661
        2019-04-30     0.200390
        2019-05-31    -0.264500
        2019-06-30     0.516568
        2019-07-31     0.803539
        2019-08-31    -0.372845
        2019-09-30     0.088650
        Freq: M, dtype: float64
```



```
In [46]: df.cumsum() ❹
```

```
Out[46]:
```

	№1	№2	№3	№4
2019-01-31	-1.749765	0.342680	1.153036	-0.252436
2019-02-28	-0.768445	0.856899	1.374215	-1.322479
2019-03-31	-0.957941	1.111901	0.916188	-0.887316
2019-04-30	-1.541536	1.928748	1.588909	-0.991727
2019-05-31	-2.072816	2.958480	1.150774	-2.110045
2019-06-30	-0.453834	4.500086	0.898895	-2.952481
2019-07-31	-0.269316	5.437168	1.629895	-1.590925
2019-08-31	-0.595554	5.492844	1.852294	-3.034142
2019-09-30	-1.351906	6.309298	2.602739	-3.490089

❶ Сумма по столбцам.

❷ Среднее по столбцам.

❸ Среднее по строкам.

❹ Накопительная сумма по столбцам (начиная с первого индекса).

К объектам `DataFrame` также можно применять универсальные функции `NumPy`.

```
In [47]: np.mean(df) ❶
```

```
Out[47]: №1    -0.150212  
        №2     0.701033  
        №3     0.289193  
        №4    -0.387788  
        dtype: float64
```

```
In [48]: np.log(df) ❷
```

```
Out[48]:
```

	№1	№2	№3	№4
2019-01-31	NaN	-1.070957	0.142398	NaN
2019-02-28	-0.018856	-0.665106	-1.508780	NaN
2019-03-31	NaN	-1.366486	NaN	-0.832033
2019-04-30	NaN	-0.202303	-0.396425	NaN
2019-05-31	NaN	0.029299	NaN	NaN
2019-06-30	0.481797	0.432824	NaN	NaN
2019-07-31	-1.690005	-0.064984	-0.313341	0.308628
2019-08-31	NaN	-2.888206	-1.503279	NaN
2019-09-30	NaN	-0.202785	-0.287089	NaN

In [49]: np.sqrt(abs(df)) ③

Out[49]:

	№1	№2	№3	№4
2019-01-31	1.322787	0.585389	1.073795	0.502430
2019-02-28	0.990616	0.717091	0.470297	1.034429
2019-03-31	0.435311	0.504977	0.676777	0.659669
2019-04-30	0.763934	0.903796	0.820196	0.323127
2019-05-31	0.728890	1.014757	0.661918	1.057506
2019-06-30	1.272392	1.241614	0.501876	0.917843
2019-07-31	0.429556	0.968030	0.854986	1.166857
2019-08-31	0.571173	0.235958	0.471593	1.201340
2019-09-30	0.869685	0.903578	0.866282	0.675238

In [50]: np.sqrt(abs(df)).sum() ④

Out[50]: №1 7.384345

№2 7.075190

№3 6.397719

№4 7.538440

dtype: float64

In [51]: 100 \* df + 100 ⑤

Out[51]:

	№1	№2	№3	№4
2019-01-31	-74.976547	134.268040	215.303580	74.756396
2019-02-28	198.132079	151.421884	122.117967	-7.004333
2019-03-31	81.050417	125.500144	54.197301	143.516349
2019-04-30	41.640495	181.684707	167.272081	89.558886
2019-05-31	46.871962	202.973269	56.186438	-11.831825
2019-06-30	261.898166	254.160517	74.812086	15.756426
2019-07-31	118.451869	193.708220	173.100034	236.155613
2019-08-31	67.376194	105.567601	122.239961	-44.321700
2019-09-30	24.364769	181.645401	175.044476	54.405307

- ① Среднее по столбцам.
- ② Вычисление натурального логарифма для каждого значения (интерпретатор выдаст предупреждение, но продолжит вычисления, и в таблице результатов появятся значения NaN).
- ③ Вычисление квадратного корня для модуля каждого значения...
- ④ ...и суммирование полученных результатов по столбцам.
- ⑤ Линейное преобразование числовых данных.



## Универсальные функции NumPy

К объекту `DataFrame` можно применить те же самые универсальные функции NumPy, что и к объекту `ndarray`, содержащему аналогичные данные.

Библиотека `pandas` достаточно терпима к ошибкам в том смысле, что она самостоятельно перехватывает их, вставляя в результирующую таблицу значение `NaN`, если соответствующую математическую операцию невозможно выполнить. Более того, во многих случаях с полученной таблицей можно работать так, как если бы она была полной. Это очень удобно, ведь на практике неполные наборы данных встречаются чаще, чем хотелось бы.

## Основные инструменты визуализации

Когда данные хранятся в объекте `DataFrame`, для построения графика, как правило, достаточно ввести одну-единственную строку кода (рис. 5.1).

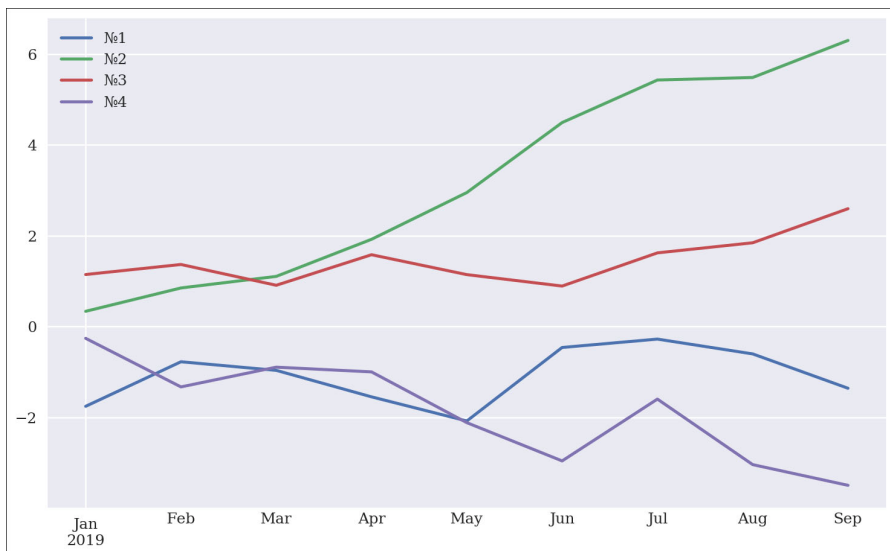


Рис. 5.1. Графики, построенные на основе данных объекта `DataFrame`

```
In [52]: from pylab import plt, mpl ❶  
         plt.style.use('seaborn') ❶  
         mpl.rcParams['font.family'] = 'serif' ❶  
         %matplotlib inline  
  
In [53]: df.cumsum().plot(lw=2.0, figsize=(10, 6)); ❷
```

- ❶ Настройка параметров графика.
- ❷ Графики накопительных сумм четырех столбцов.

В библиотеке `pandas` реализована оболочка к пакету `matplotlib` (рассматривается в главе 7), специально предназначенная для работы с объектами `DataFrame`. Основные параметры метода `plot()` описаны в табл. 5.4.

Таблица 5.4. Основные параметры метода `plot()`

Параметр	Формат	Описание
<code>x</code>	Подпись/позиция, по умолчанию <code>None</code>	Используется, если значения столбца содержат подписи для делений оси <code>X</code>
<code>y</code>	Подпись/позиция, по умолчанию <code>None</code>	Используется, если значения столбца содержат подписи для делений оси <code>Y</code>
<code>subplots</code>	Булево значение, по умолчанию <code>False</code>	Представление столбцов в отдельных областях графика
<code>sharex</code>	Булево значение, по умолчанию <code>True</code>	Общая ось <code>X</code>
<code>sharey</code>	Булево значение, по умолчанию <code>False</code>	Общая ось <code>Y</code>
<code>use_index</code>	Булево значение, по умолчанию <code>True</code>	Применение индексов объекта <code>DataFrame</code> в качестве делений оси <code>X</code>
<code>stacked</code>	Булево значение, по умолчанию <code>False</code>	Представление значений с накоплением (только для гистограмм)
<code>sort_columns</code>	Булево значение, по умолчанию <code>False</code>	Предварительная сортировка значений столбцов
<code>title</code>	Строка, по умолчанию <code>None</code>	Название графика
<code>grid</code>	Булево значение, по умолчанию <code>False</code>	Отображение координатной сетки
<code>legend</code>	Булево значение, по умолчанию <code>True</code>	Отображение легенды
<code>ax</code>	Объект оси <code>matplotlib</code>	Объект <code>matplotlib</code> , определяющий ось, вдоль которой строится график
<code>style</code>	Строка или список/словарь	Стиль линии (отдельно для каждого столбца)
<code>kind</code>	Строка (например, <code>"line"</code> , <code>"bar"</code> , <code>"barh"</code> , <code>"kde"</code> , <code>"density"</code> )	Тип графика
<code>logx</code>	Булево значение, по умолчанию <code>False</code>	Логарифмический масштаб оси <code>X</code>
<code>logy</code>	Булево значение, по умолчанию <code>False</code>	Логарифмический масштаб оси <code>Y</code>
<code>xticks</code>	Последовательность, по умолчанию <code>Index</code>	Деления оси <code>X</code>
<code>yticks</code>	Последовательность, по умолчанию <code>Values</code>	Деления оси <code>Y</code>

Параметр	Формат	Описание
xlim	Двухэлементный кортеж, список	Границы области построения для оси X
ylim	Двухэлементный кортеж, список	Границы области построения для оси Y
rot	Целое число, по умолчанию None	Поворот делений оси X
secondary_y	Булево значение/последовательность, по умолчанию False	Построение на вторичной оси Y
mark_right	Булево значение, по умолчанию True	Автоматическое обозначение вторичной оси
colormap	Строка/объект colormap, по умолчанию None	Цветовая карта графиков
kws	Ключевые слова	Параметры, передаваемые библиотеке matplotlib

В качестве примера построим гистограмму того же набора данных (рис. 5.2).

```
In [54]: df.plot(figsize=(10, 6), rot=15); ❶
         # df.plot(kind='bar', figsize=(10, 6)) ❷
```

- ❶ Построение гистограммы с помощью метода `plot.bar()`.
- ❷ Альтернативный синтаксис: указание типа диаграммы (параметр `kind`) в методе `plot()`.

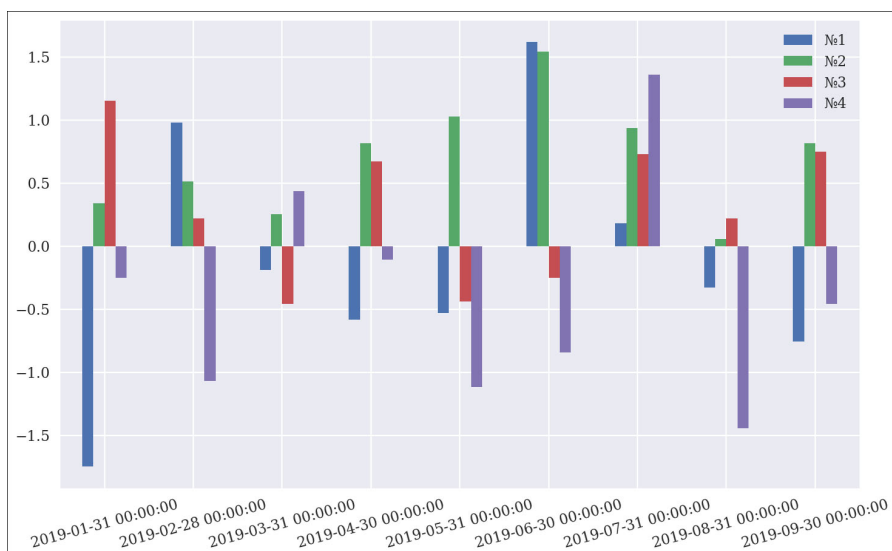


Рис. 5.2. Гистограмма, построенная на основе данных объекта `DataFrame`

# Класс Series

Помимо класса `DataFrame` в библиотеке `pandas` имеется другой важный класс: `Series`. Он применяется для хранения данных, представленных единственным столбцом. В этом смысле класс `Series` можно рассматривать как частный случай класса `DataFrame`. Объект `Series` будет получен при выборе отдельного столбца из табличной структуры объекта `DataFrame`.

```
In [55]: type(df)
```

```
Out[55]: pandas.core.frame.DataFrame
```

```
In [56]: S = pd.Series(np.linspace(0, 15, 7), name='series')
```

```
In [57]: S
```

```
Out[57]: 0      0.0
         1      2.5
         2      5.0
         3      7.5
         4     10.0
         5     12.5
         6     15.0
         Name: series, dtype: float64
```

```
In [58]: type(S)
```

```
Out[58]: pandas.core.series.Series
```

```
In [59]: s = df['№1']
```

```
In [60]: s
```

```
Out[60]: 2019-01-31   -1.749765
         2019-02-28    0.981321
         2019-03-31   -0.189496
         2019-04-30   -0.583595
         2019-05-31   -0.531280
         2019-06-30    1.618982
         2019-07-31    0.184519
         2019-08-31   -0.326238
         2019-09-30   -0.756352
         Freq: M, Name: №1, dtype: float64
```

```
In [61]: type(s)
```

```
Out[61]: pandas.core.series.Series
```

Все основные методы объекта `DataFrame` применимы и к объекту `Series`. В качестве иллюстрации рассмотрим применение методов `mean()` и `plot()` (рис. 5.3).

```
In [62]: s.mean()
```

```
Out[62]: -0.15021177307319458
```

```
In [63]: s.plot(lw=2.0, figsize=(10, 6));
```

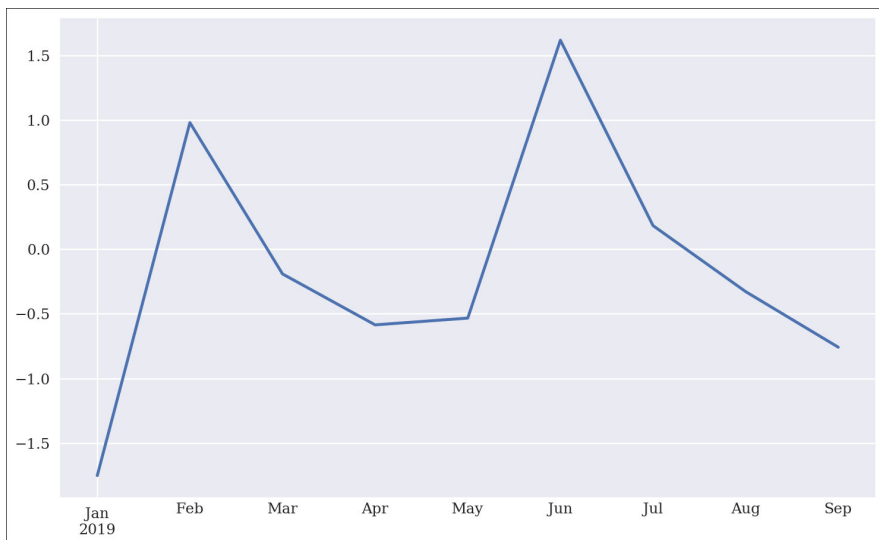


Рис. 5.3. График данных, хранящихся в объекте `Series`

## Группирование данных

В библиотеке `pandas` поддерживаются гибкие возможности группирования данных подобно тому, как это реализовано в SQL и сводных таблицах Microsoft Excel. Например, в объект `DataFrame` можно добавить столбец, обозначающий квартал, по которому будет выполняться группировка.

```
In [64]: df['Квартал'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2',  
                        'Q2', 'Q3', 'Q3', 'Q3']
```

```
df
```

```
Out[64]:
```

	№1	№2	№3	№4	Квартал
2019-01-31	-1.749765	0.342680	1.153036	-0.252436	Q1
2019-02-28	0.981321	0.514219	0.221180	-1.070043	Q1

<b>2019-03-31</b>	-0.189496	0.255001	-0.458027	0.435163	Q1
<b>2019-04-30</b>	-0.583595	0.816847	0.672721	-0.104411	Q2
<b>2019-05-31</b>	-0.531280	1.029733	-0.438136	-1.118318	Q2
<b>2019-06-30</b>	1.618982	1.541605	-0.251879	-0.842436	Q2
<b>2019-07-31</b>	0.184519	0.937082	0.731000	1.361556	Q3
<b>2019-08-31</b>	-0.326238	0.055676	0.222400	-1.443217	Q3
<b>2019-09-30</b>	-0.756352	0.816454	0.750445	-0.455947	Q3

Вот как сгруппировать данные по кварталам и вычислить статистику по ним.

In [65]: `groups = df.groupby('Квартал')` ❶

In [66]: `groups.size()` ❷

Out[66]: Квартал

Q1	3
Q2	3
Q3	3

dtype: int64

In [67]: `groups.mean()` ❸

Out[67]:

	№1	№2	№3	№4
<b>Квартал</b>				
<b>Q1</b>	-0.319314	0.370634	0.305396	-0.295772
<b>Q2</b>	0.168035	1.129395	-0.005765	-0.688388
<b>Q3</b>	-0.299357	0.603071	0.567948	-0.179203

In [68]: `groups.max()` ❹

Out[68]:

	№1	№2	№3	№4
<b>Квартал</b>				
<b>Q1</b>	0.981321	0.514219	1.153036	0.435163
<b>Q2</b>	1.618982	1.541605	0.672721	-0.104411
<b>Q3</b>	0.184519	0.937082	0.750445	1.361556

In [69]: `groups.aggregate([min, max]).round(2)` ❺

Out[69]:

	№1		№2		№3		№4	
	min	max	min	max	min	max	min	max
<b>Квартал</b>								
<b>Q1</b>	-1.75	0.98	0.26	0.51	-0.46	1.15	-1.07	0.44
<b>Q2</b>	-0.58	1.62	0.82	1.54	-0.44	0.67	-1.12	-0.10
<b>Q3</b>	-0.76	0.18	0.06	0.94	0.22	0.75	-1.44	1.36



- ❶ Группировка по столбцу Квартал.
- ❷ Подсчет количества строк в каждой группе.
- ❸ Среднее по каждому столбцу.
- ❹ Максимальное значение по каждому столбцу.
- ❺ Минимальное и максимальное значения по каждому столбцу.

Группировку можно выполнять сразу по нескольким столбцам. Для примера добавим в таблицу еще один столбец, в котором каждый месяц индекса будет помечаться как четный или нечетный.

```
In [70]: df['Чет/нечет'] = ['Нечет', 'Чет', 'Нечет', 'Чет', 'Нечет',
                             'Чет', 'Нечет', 'Чет', 'Нечет']
```

```
In [71]: groups = df.groupby(['Квартал', 'Чет/нечет'])
```

```
In [72]: groups.size()
```

```
Out[72]: Квартал  Чет/нечет
Q1          Нечет      2
          Чет        1
Q2          Нечет      1
          Чет        2
Q3          Нечет      2
          Чет        1
dtype: int64
```

```
In [73]: groups[['№1', '№4']].aggregate([sum, np.mean])
```

```
Out[73]:
```

Квартал	Чет/нечет	№1		№4	
		sum	mean	sum	mean
Q1	Нечет	-1.939261	-0.969631	0.182727	0.091364
	Чет	0.981321	0.981321	-1.070043	-1.070043
Q2	Нечет	-0.531280	-0.531280	-1.118318	-1.118318
	Чет	1.035387	0.517693	-0.946847	-0.473423
Q3	Нечет	-0.571834	-0.285917	0.905609	0.452805
	Чет	-0.326238	-0.326238	-1.443217	-1.443217

## Сложные операции извлечения данных

Чаще всего данные извлекаются согласно условиям, накладываемым на значения столбцов. В качестве примера рассмотрим следующий набор данных.

```
In [74]: data = np.random.standard_normal((10, 2)) ❶
```

```
In [75]: df = pd.DataFrame(data, columns=['x', 'y']) ❷
```

```
In [76]: df.info() ❸
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 2 columns):
x      10 non-null float64
y      10 non-null float64
dtypes: float64(2)
memory usage: 240.0 bytes
```

```
In [77]: df.head() ❹
```

Out[77]:

	x	y
0	1.189622	-1.690617
1	-1.356399	-1.232435
2	-0.544439	-0.668172
3	0.007315	-0.612939
4	1.299748	-1.733096

```
In [78]: df.tail() ❺
```

Out[78]:

	x	y
5	-0.983310	0.357508
6	-1.613579	1.470714
7	-1.188018	-0.549746
8	-0.940046	-0.827932
9	0.108863	0.507810

- ❶ Массив `ndarray` случайных чисел с нормальным распределением.
- ❷ Объект `DataFrame`, содержащий тот же набор случайных чисел.
- ❸ Первые пять строк, возвращаемые методом `head()`.
- ❹ Последние пять строк, возвращаемые методом `tail()`.

В следующем примере показано, как применять логические операции и операции сравнения к значениям двух столбцов.

```
In [79]: df['x'] > 0.5 ❶
```

```
Out[79]: 0    True
         1    False
         2    False
         3    False
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         Name: x, dtype: bool
```

```
In [80]: (df['x'] > 0) & (df['y'] < 0) ❷
```

```
Out[80]: 0    True
         1    False
         2    False
         3     True
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         dtype: bool
```

```
In [81]: (df['x'] > 0) | (df['y'] < 0) ❸
```

```
Out[81]: 0    True
         1     True
         2     True
         3     True
         4     True
         5    False
         6    False
         7     True
         8     True
         9     True
         dtype: bool
```

❶ Превышают ли элементы столбца x значение 0.5?

- ❷ Являются ли значения столбца *x* положительными *и* значения столбца *y* отрицательными?
- ❸ Являются ли значения столбца *x* положительными *или* значения столбца *y* отрицательными?

Полученные в предыдущем примере объекты *Series* с булевыми значениями удобно использовать в качестве критериев отбора данных. Альтернативным решением будет извлечение данных с помощью метода `query()`, в котором критерий отбора задается в виде строки.

```
In [82]: df[df['x'] > 0] ❶
```

```
Out[82]:
```

	x	y
0	1.189622	-1.690617
3	0.007315	-0.612939
4	1.299748	-1.733096
9	0.108863	0.507810

```
In [83]: df.query('x > 0') ❶
```

```
Out[83]:
```

	x	y
0	1.189622	-1.690617
3	0.007315	-0.612939
4	1.299748	-1.733096
9	0.108863	0.507810

```
In [84]: df[(df['x'] > 0) & (df['y'] < 0)] ❷
```

```
Out[84]:
```

	x	y
0	1.189622	-1.690617
3	0.007315	-0.612939
4	1.299748	-1.733096

```
In [85]: df.query('x > 0 & y < 0') ❷
```

```
Out[85]:
```

	x	y
0	1.189622	-1.690617
3	0.007315	-0.612939
4	1.299748	-1.733096

```
In [86]: df[(df.x > 0) | (df.y < 0)] ❸
```

```
Out[86]:
```

	x	y
0	1.189622	-1.690617
1	-1.356399	-1.232435
2	-0.544439	-0.668172
3	0.007315	-0.612939
4	1.299748	-1.733096
7	-1.188018	-0.549746
8	-0.940046	-0.827932
9	0.108863	0.507810

- ❶ Строки, в которых значения столбца x больше 0.
- ❷ Строки, в которых значения столбца x положительные *и* значения столбца y отрицательные.
- ❸ Строки, в которых значения столбца x положительные *или* значения столбца y отрицательные (в данном случае доступ к столбцам осуществляется через соответствующие атрибуты).

Операторы сравнения могут также применяться сразу ко всему объекту DataFrame.

In [87]: df > 0 ❶

Out[87]:

	x	y
0	True	False
1	False	False
2	False	False
3	True	False
4	True	False
5	False	True
6	False	True
7	False	False
8	False	False
9	True	True

In [88]: df[df > 0] ❷

Out[88]:

	x	y
0	1.189622	NaN
1	NaN	NaN
2	NaN	NaN
3	0.007315	NaN
4	1.299748	NaN
5	NaN	0.357508

6	NaN	1.470714
7	NaN	NaN
8	NaN	NaN
9	0.108863	0.507810

- 1 Определение того, какие из значений объекта `DataFrame` положительные.
- 2 Выбор положительных чисел и замена всех остальных чисел значением `NaN`.

## Конкатенация, соединение и слияние данных

В этом разделе мы рассмотрим различные подходы к объединению двух простых наборов данных, хранящихся в объектах `DataFrame`. Во всех примерах используются следующие наборы.

```
In [89]: df1 = pd.DataFrame(['100', '200', '300', '400'],
                             index=['a', 'b', 'c', 'd'],
                             columns=['A',])
```

```
In [90]: df1
Out[90]:
```

	A
a	100
b	200
c	300
d	400

```
In [91]: df2 = pd.DataFrame(['200', '150', '50'],
                             index=['f', 'b', 'd'],
                             columns=['B',])
```

```
In [92]: df2
Out[92]:
```

	B
f	200
b	150
d	50

## Конкатенация

*Конкатенация*, или *присоединение*, — это операция добавления строк из одного объекта `DataFrame` в другой. Такая операция реализуется с помощью

метода `append()` или функции `pd.concat()`. Основное внимание следует обратить на обработку индексов.

In [93]: `df1.append(df2, sort=False)` ❶

Out[93]:

	A	B
a	100	NaN
b	200	NaN
c	300	NaN
d	400	NaN
f	NaN	200
b	NaN	150
d	NaN	50

In [94]: `df1.append(df2, ignore_index=True, sort=False)` ❷

Out[94]:

	A	B
0	100	NaN
1	200	NaN
2	300	NaN
3	400	NaN
4	NaN	200
5	NaN	150
6	NaN	50

In [95]: `pd.concat((df1, df2), sort=False)` ❸

Out[95]:

	A	B
a	100	NaN
b	200	NaN
c	300	NaN
d	400	NaN
f	NaN	200
b	NaN	150
d	NaN	50

In [96]: `pd.concat((df1, df2), ignore_index=True, sort=False)` ❹

Out[96]:

	A	B
0	100	NaN
1	200	NaN
2	300	NaN
3	400	NaN
4	NaN	200

```
5 NaN 150
6 NaN 50
```

- ❶ Добавление содержимого объекта `df2` в объект `df1` в виде новых строк.
- ❷ То же самое, но без учета индексов.
- ❸ Тот же результат, что и в п. 1.
- ❹ Тот же результат, что и в п. 2.

## Соединение

При соединении двух наборов направление копирования записей имеет значение. По умолчанию используются только индексы первого объекта `DataFrame`. Такого рода операция называется *левым соединением* (`left join`).

```
In [97]: df1.join(df2) ❶
Out[97]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50

```
In [98]: df2.join(df1) ❷
Out[98]:
```

	B	A
f	200	NaN
b	150	200
d	50	400

- ❶ Применяются индексы объекта `df1`.
- ❷ Применяются индексы объекта `df2`.

Метод `join()` поддерживает четыре способа соединения, которые различаются способом обработки индексов и приводят к разным результатам.

```
In [99]: df1.join(df2, how='left') ❶
Out[99]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50



```
In [100]: df1.join(df2, how='right') ❷
```

```
Out[100]:
```

	A	B
f	NaN	200
b	200	150
d	400	50

```
In [101]: df1.join(df2, how='inner') ❸
```

```
Out[101]:
```

	A	B
b	200	150
d	400	50

```
In [102]: df1.join(df2, how='outer') ❹
```

```
Out[102]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50
f	NaN	200

- ❶ Левое соединение (операция по умолчанию).
- ❷ Правое соединение равнозначно соединению объектов `DataFrame` в обратном порядке.
- ❸ При внутреннем соединении отбираются только строки с совпадающими индексами.
- ❹ При внешнем соединении сохраняются все индексы.

Допускается соединение с пустым объектом `DataFrame`. В таком случае столбцы создаются *в порядке указания*, что напоминает левое соединение.

```
In [103]: df = pd.DataFrame()
```

```
In [104]: df['A'] = df1['A'] ❶
```

```
In [105]: df
```

```
Out[105]:
```

	A
a	100
b	200

```
c 300
d 400
```

```
In [106]: df['B'] = df2 ❷
```

```
In [107]: df
Out[107]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50

❶ Объект df1 становится столбцом A.

❷ Объект df2 становится столбцом B.

Использование словаря для объединения наборов данных приводит к результату, аналогичному внешнему соединению, поскольку в таком случае столбцы создаются *одновременно*.

```
In [108]: df = pd.DataFrame({'A': df1['A'], 'B': df2['B']}) ❶
```

```
In [109]: df
Out[109]:
```

	A	B
a	100	NaN
b	200	150
c	300	NaN
d	400	50
f	NaN	200

❶ Столбцы объектов DataFrame используются в качестве значений словаря.

## Слияние

Если при соединении учитываются индексы объектов DataFrame, то при слиянии учитываются совпадающие столбцы. Чтобы продемонстрировать это, добавим в оба исходных объекта DataFrame новый столбец C.

```
In [110]: c = pd.Series([250, 150, 50], index=['b', 'd', 'c'])
          df1['C'] = c
          df2['C'] = c
```

```
In [111]: df1
```

```
Out[111]:
```

	A	C
a	100	NaN
b	200	250.0
c	300	50.0
d	400	150.0

```
In [112]: df2
```

```
Out[112]:
```

	B	C
f	200	NaN
b	150	250.0
d	50	150.0

По умолчанию операция слияния выполняется по единственному общему столбцу C. Но есть и другие варианты, например *внешнее слияние*.

```
In [113]: pd.merge(df1, df2) ❶
```

```
Out[113]:
```

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	400	150.0	50

```
In [114]: pd.merge(df1, df2, on='C') ❶
```

```
Out[114]:
```

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	400	150.0	50

```
In [115]: pd.merge(df1, df2, how='outer') ❷
```

```
Out[115]:
```

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	300	50.0	NaN
3	400	150.0	50

❶ Слияние по столбцу C (вариант по умолчанию).

❷ Внешнее слияние, позволяющее сохранить все строки.

Другие способы слияния таблиц продемонстрированы ниже.

```
In [116]: pd.merge(df1, df2, left_on='A', right_on='B')
```

```
Out[116]:
```

	A	C_x	B	C_y
0	200	250.0	200	NaN

```
In [117]: pd.merge(df1, df2, left_on='A', right_on='B', how='outer')
```

```
Out[117]:
```

	A	C_x	B	C_y
0	100	NaN	NaN	NaN
1	200	250.0	200	NaN
2	300	50.0	NaN	NaN
3	400	150.0	NaN	NaN
4	NaN	NaN	150	250.0
5	NaN	NaN	50	150.0

```
In [118]: pd.merge(df1, df2, left_index=True, right_index=True)
```

```
Out[118]:
```

	A	C_x	B	C_y
b	200	250.0	150	250.0
d	400	150.0	50	150.0

```
In [119]: pd.merge(df1, df2, on='C', left_index=True)
```

```
Out[119]:
```

	A	C	B
f	100	NaN	200
b	200	250.0	150
d	400	150.0	50

```
In [120]: pd.merge(df1, df2, on='C', right_index=True)
```

```
Out[120]:
```

	A	C	B
a	100	NaN	200
b	200	250.0	150
d	400	150.0	50

```
In [121]: pd.merge(df1, df2, on='C', left_index=True,  
                  right_index=True)
```

```
Out[121]:
```

	A	C	B
b	200	250.0	150
d	400	150.0	50

## Производительность вычислений

Из многочисленных примеров, приведенных в этой главе, прекрасно видно, что в библиотеке `pandas` одну и ту же задачу можно решить несколькими способами. В данном разделе мы сравним производительность поэлементного суммирования двух столбцов. Сначала сгенерируем набор данных средствами библиотеки `NumPy`.

```
In [122]: data = np.random.standard_normal((1000000, 2)) ❶
```

```
In [123]: data.nbytes ❶
```

```
Out[123]: 16000000
```

```
In [124]: df = pd.DataFrame(data, columns=['x', 'y']) ❷
```

```
In [125]: df.info() ❷
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
x      1000000 non-null float64
y      1000000 non-null float64
dtypes: float64(2)
memory usage: 15.3 MB
```

❶ Объект `ndarray`, хранящий случайные числа.

❷ Объект `DataFrame`, хранящий случайные числа.

Далее замерим производительность операций суммирования.

```
In [126]: %time res = df['x'] + df['y'] ❶
```

```
CPU times: user 7.35 ms, sys: 7.43 ms, total: 14.8 ms
Wall time: 7.48 ms
```

```
In [127]: res[:3]
```

```
Out[127]: 0      0.387242
          1     -0.969343
          2     -0.863159
          dtype: float64
```

```
In [128]: %time res = df.sum(axis=1) ❷
```

```
CPU times: user 130 ms, sys: 30.6 ms, total: 161 ms
Wall time: 101 ms
```

```
In [129]: res[:3]
Out[129]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [130]: %time res = df.values.sum(axis=1) ❸
CPU times: user 50.3 ms, sys: 2.75 ms, total: 53.1 ms
Wall time: 27.9 ms
```

```
In [131]: res[:3]
Out[131]: array([ 0.3872424 , -0.96934273, -0.86315944])
```

```
In [132]: %time res = np.sum(df, axis=1) ❹
CPU times: user 127 ms, sys: 15.1 ms, total: 142 ms
Wall time: 73.7 ms
```

```
In [133]: res[:3]
Out[133]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [134]: %time res = np.sum(df.values, axis=1) ❺
CPU times: user 49.3 ms, sys: 2.36 ms, total: 51.7 ms
Wall time: 26.9 ms
```

```
In [135]: res[:3]
Out[135]: array([ 0.3872424 , -0.96934273, -0.86315944])
```

- ❶ Непосредственная работа со столбцами (объектами Series) — самый быстрый подход.
- ❷ Вычисление сумм с помощью метода `sum()` объекта `DataFrame`.
- ❸ Вычисление сумм с помощью метода `sum()` объекта `ndarray`.
- ❹ Вычисление сумм путем применения функции `np.sum()` к объекту `DataFrame`.
- ❺ Вычисление сумм путем применения функции `np.sum()` к объекту `ndarray`.

Кроме того, значения можно суммировать с помощью методов `eval()` и `apply()`<sup>1</sup>.

```
In [136]: %time res = df.eval('x + y') ❶
CPU times: user 25.5 ms, sys: 17.7 ms, total: 43.2 ms
Wall time: 22.5 ms
```

```
In [137]: res[:3]
Out[137]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

```
In [138]: %time res = df.apply(lambda row: row['x'] + row['y'],
                                axis=1) ❷
CPU times: user 19.6 s, sys: 83.3 ms, total: 19.7 s
Wall time: 19.9 s
```

```
In [139]: res[:3]
Out[139]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

- ❶ Метод `eval()` применяется для вычисления сложных математических выражений; имена столбцов извлекаются из выражения.
- ❷ Метод `apply()` выполняет суммирование построчно, а потому оказывается самым медленным; это эквивалентно выполнению цикла по всем строкам в коде Python.



### Выбирайте с умом

В библиотеке `pandas` одну и ту же задачу можно решить множеством способов. Выбирая самый подходящий вариант, обращайте внимание на производительность кода, если скорость работы приложения имеет для вас значение. Даже в нашем простом примере скорость работы некоторых методов различается на порядок.

---

<sup>1</sup> Применение метода `eval()` требует наличия пакета `numexpr` (<https://numexpr.readthedocs.io/en/latest/>).

## Резюме

Библиотека `pandas` — мощный инструмент анализа данных и центральный пакет так называемого *стека PyData*. Ее класс `DataFrame` специально предназначен для работы с табличными данными любого рода. Большинство операций с такими объектами векторизовано, что позволяет не только уменьшить объем кода, но и повысить его производительность. Кроме того, библиотека `pandas` (в отличие, например, от `NumPy`) позволяет работать с неполными наборами данных. Как следствие, библиотека `pandas` и ее класс `DataFrame` будут составлять основу многих решений, описанных в последующих главах, где мы будем рассматривать другие инструменты библиотеки по мере необходимости.

## Дополнительные ресурсы

Библиотека `pandas` — это проект с открытым исходным кодом, снабженный подробной документацией, которая доступна как в виде онлайн-ресурса, так и в виде загружаемого PDF-файла<sup>2</sup> (<http://pandas.pydata.org/>).

Как и в случае с `NumPy`, для знакомства с библиотекой `pandas` подойдут следующие книги.

- McKinney, Wes. *Python for Data Analysis* (2017, O'Reilly).
- VanderPlas, Jake. *Python Data Science Handbook* (2016, O'Reilly).

---

<sup>2</sup> На момент написания книги PDF-версия документации насчитывала более 2500 страниц.





---

# Объектно-ориентированное программирование

Разработчики программного обеспечения должны контролировать сложность, а не создавать ее.

*Памела Зейв*

Объектно-ориентированное программирование (ООП) — одна из самых популярных парадигм программирования. При правильном применении ООП дает существенные преимущества по сравнению, например, с процедурным программированием. С точки зрения финансового моделирования и реализации финансовых алгоритмов ООП — оптимальный выбор. Тем не менее у этой парадигмы немало критиков, выражающих свой скептицизм как в отношении отдельных ее аспектов, так и всей концепции в целом. В этой главе мы будем придерживаться нейтральной точки зрения, считая ООП достаточно эффективной, хотя и не единственно возможной методологией решения большинства финансовых задач.

При знакомстве с ООП следует изучить основную терминологию.

## *Класс*

Абстрактное определение конкретного типа объектов, например человека.

## *Объект*

Экземпляр класса, например Сандра.

## *Атрибут*

Характеристика класса (*атрибут класса*) или экземпляра класса (*атрибут объекта*). Например, характеристикой человека может быть пол (мужской или женский) или цвет глаз.

## *Метод*

Действие, выполняемое классом или экземпляром класса, например ходьба.

## Параметры

Аргументы, передаваемые методам, например количество шагов.

## Инициализация

Процесс создания конкретного объекта на основе абстрактного класса.

В переводе на язык Python простой класс `HumanBeing`, описывающий человека, может выглядеть так.

```
In [1]: class HumanBeing(object): ❶
        def __init__(self, first_name, eye_color): ❷
            self.first_name = first_name ❸
            self.eye_color = eye_color ❹
            self.position = 0 ❺
        def walk_steps(self, steps): ❻
            self.position += steps ❼
```

- ❶ Инструкция объявления класса.
- ❷ Специальный метод, вызываемый при создании экземпляра класса; ключевое слово `self` ссылается на текущий экземпляр класса.
- ❸ Атрибут имени, инициализируемый значением первого параметра.
- ❹ Атрибут цвета глаз, инициализируемый значением второго параметра.
- ❺ Атрибут позиции инициализируется значением 0.
- ❻ Определение метода, реализующего процесс ходьбы; в качестве параметра указывается количество шагов.
- ❼ Код, меняющий позицию на заданное число шагов.

На основе класса создаются объекты-экземпляры.

```
In [2]: Sandra = HumanBeing('Сандра', 'голубой') ❶
```

```
In [3]: Sandra.first_name ❷
Out[3]: 'Сандра'
```

```
In [4]: Sandra.position ❷
Out[4]: 0
```

```
In [5]: Sandra.walk_steps(5) ❸
```

```
In [6]: Sandra.position ❹
Out[6]: 5
```

- ❶ Создание экземпляра класса.
- ❷ Получение значений атрибутов.
- ❸ Вызов метода.
- ❹ Получение обновленного значения атрибута `position`.

С точки зрения удобства применения ООП дает следующие преимущества.

#### *Понятный способ мышления*

Мы привыкли мыслить категориями реальных или абстрактных объектов, таких как автомобиль или финансовый инструмент. ООП идеально подходит для моделирования таких объектов и их характеристик.

#### *Упрощение кода*

ООП помогает уменьшить сложность решаемых задач и реализуемых алгоритмов, выполняя моделирование на уровне признаков.

#### *Удобный программный интерфейс*

ООП позволяет создавать удобные программные интерфейсы и в целом получать более компактный код. Наглядными примерами служат классы `ndarray` библиотеки `NumPy` и `DataFrame` библиотеки `pandas`.

#### *Естественный для Python способ моделирования*

Как бы там ни было, ООП является основной парадигмой программирования на Python. Отсюда проистекает выражение “в Python все что угодно является объектом”. ООП позволяет создавать пользовательские классы, экземпляры которых ведут себя подобно стандартным объектам Python.

В то же время ООП дает и ряд *технических преимуществ*.

#### *Абстракция*

Наличие атрибутов и методов позволяет создавать абстрактные, гибкие модели объектов, позволяющие сконцентрироваться только на том, что важно, отбросив все остальное. В финансовых приложениях это может означать наличие универсального класса, моделирующего абстрактный финансовый инструмент. Экземплярами такого класса будут конкретные финансовые продукты, разработанные, например, неким инвестиционным банком.

#### *Модульность*

Концепция ООП предусматривает разбивку кода на модули, подключаемые к приложениям по мере необходимости. Например, европейский

опцион на акции можно смоделировать с помощью одного класса или двух классов: один — для представления акций, другой — для самого опциона.

### *Наследование*

Наследование — это концепция ООП, согласно которой класс может заимствовать атрибуты и методы у другого класса. В финансовом приложении от общего класса финансовых инструментов может быть унаследован класс деривативов, от него — класс европейских опционов, а от последнего — класс европейского колл-опциона. Каждый класс наследует атрибуты и методы всех классов более высокого уровня.

### *Агрегирование*

При агрегировании объект как минимум частично состоит из других объектов, которые могут существовать независимо. Например, атрибутами класса европейского колл-опциона могут быть объекты, представляющие базовые акции и краткосрочные процентные ставки на их размещение. При этом данные объекты могут независимо использоваться любыми другими объектами.

### *Композиция*

Композиция подобна агрегированию, только в данном случае объекты не являются независимыми. В качестве примера рассмотрим процентный своп с фиксированной и плавающей ставками. Объекты ставок не могут существовать отдельно от объекта самого свопа.

### *Полиморфизм*

Полиморфизм бывает разных видов. В частности, в Python широко применяется *неявная типизация* (duck typing), при которой стандартные операции можно выполнять по отношению к самым разным классам и их экземплярам, не располагая явной информацией об их типе. Для класса финансовых инструментов это означает возможность вызова, например, метода `get_current_price()` независимо от типа объекта (акция, опцион, процентный своп).

### *Инкапсуляция*

Концепция инкапсуляции предполагает предоставление доступа к данным класса только через открытые методы. Например, у класса, моделирующего акцию, может быть атрибут `current_stock_price`. Инкапсуляция означает, что доступ к данному атрибуту можно получить только с помощью метода `get_current_stock_price()`, а сам атрибут

скрыт от пользователя. Такой подход позволяет избежать побочных эффектов, связанных с непосредственным изменением атрибутов. В то же время в Python возможности инкапсуляции ограничены.

На более высоком уровне преимущества ООП можно свести к *двум общим целям* программной инженерии.

#### *Повторное использование кода*

Наследование и полиморфизм делают возможным повторное использование кода, что способствует повышению эффективности разработки программного обеспечения. Кроме того, они упрощают последующее сопровождение программного продукта.

#### *Отсутствие избыточности*

Вместе с тем ООП позволяет писать код, практически лишенный избыточности. Не приходится тратить усилия на повторную реализацию алгоритмов, плюс сокращается время, затрачиваемое на отладку, тестирование и сопровождение программ.

В главе рассматриваются следующие темы.

#### *Обзор объектов Python*

В этом разделе рассматриваются уже известные вам объекты Python в контексте ООП.

#### *Основные операции с классами Python*

В этом разделе мы применим ООП в контексте работы с финансовыми инструментами и инвестиционными портфелями.

#### *Модель данных Python*

В этом разделе мы поговорим о модели данных Python и важности специальных методов.

## Обзор объектов Python

Знакомство с ООП лучше всего начать с рассмотрения стандартных объектов, уже известных вам по предыдущим главам.

### **int**

Начнем с объекта `int` (целое число). Заметьте, что даже такой простой объект обладает всеми необходимыми функциональными характеристиками.

```
In [7]: n = 5 ❶
```

```
In [8]: type(n) ❷  
Out[8]: int
```

```
In [9]: n.numerator ❸  
Out[9]: 5
```

```
In [10]: n.bit_length() ❹  
Out[10]: 3
```

```
In [11]: n + n ❺  
Out[11]: 10
```

```
In [12]: 2 * n ❻  
Out[12]: 10
```

```
In [13]: n.__sizeof__() ❼  
Out[13]: 28
```

- ❶ Новый экземпляр `n`.
- ❷ Тип объекта.
- ❸ Атрибут.
- ❹ Метод.
- ❺ Применение оператора `+` (сложение).
- ❻ Применение оператора `*` (умножение).
- ❼ Вызов специального метода `__sizeof__()` для определения объема занимаемой объектом памяти<sup>1</sup>.

## list

Списки — тоже объекты, только по сравнению с целыми числами они поддерживают большее количество методов.

```
In [14]: l = [1, 2, 3, 4] ❶
```

```
In [15]: type(l) ❷
```

---

<sup>1</sup> В Python специальные атрибуты и методы обрамляются символами двойного подчеркивания. В частности, метод `n.__sizeof__()` возвращает размер объекта `n` в байтах.

```
Out[15]: list
```

```
In [16]: l[0] ❸
```

```
Out[16]: 1
```

```
In [17]: l.append(10) ❹
```

```
In [18]: l + l ❺
```

```
Out[18]: [1, 2, 3, 4, 10, 1, 2, 3, 4, 10]
```

```
In [19]: 2 * l ❻
```

```
Out[19]: [1, 2, 3, 4, 10, 1, 2, 3, 4, 10]
```

```
In [20]: sum(l) ❼
```

```
Out[20]: 20
```

```
In [21]: l.__sizeof__() ❽
```

```
Out[21]: 104
```

- ❶ Новый экземпляр `l`.
- ❷ Тип объекта.
- ❸ Выбор элемента по индексу.
- ❹ Метод.
- ❺ Применение оператора `+` (сложение).
- ❻ Применение оператора `*` (умножение).
- ❼ Вызов специального метода `__sizeof__()` для определения объема занимаемой объектом памяти.

## **ndarray**

Если списки и целые числа — это стандартные объекты Python, то объект `ndarray` заимствуется из дополнительной библиотеки NumPy.

```
In [22]: import numpy as np ❶
```

```
In [23]: a = np.arange(16).reshape((4, 4)) ❷
```

```
In [24]: a ❸
```

```
Out[24]: array([[ 0,  1,  2,  3],  
               [ 4,  5,  6,  7],  
               [ 8,  9, 10, 11],  
               [12, 13, 14, 15]])
```



```
[ 8,  9, 10, 11],  
[12, 13, 14, 15]])
```

```
In [25]: type(a) ❸  
Out[25]: numpy.ndarray
```

- ❶ Импорт библиотеки NumPy.
- ❷ Новый экземпляр `a`.
- ❸ Тип объекта.

Несмотря на то что `ndarray` не относится к стандартным объектам, он во многих случаях ведет себя подобно таким объектам благодаря принятой в Python модели данных (рассматривается далее).

```
In [26]: a.nbytes ❶  
Out[26]: 128
```

```
In [27]: a.sum() ❷  
Out[27]: 120
```

```
In [28]: a.cumsum(axis=0) ❸  
Out[28]: array([[ 0,  1,  2,  3],  
                [ 4,  6,  8, 10],  
                [12, 15, 18, 21],  
                [24, 28, 32, 36]])
```

```
In [29]: a + a ❹  
Out[29]: array([[ 0,  2,  4,  6],  
                [ 8, 10, 12, 14],  
                [16, 18, 20, 22],  
                [24, 26, 28, 30]])
```

```
In [30]: 2 * a ❺  
Out[30]: array([[ 0,  2,  4,  6],  
                [ 8, 10, 12, 14],  
                [16, 18, 20, 22],  
                [24, 26, 28, 30]])
```

```
In [31]: sum(a) ❻  
Out[31]: array([24, 28, 32, 36])
```

```
In [32]: np.sum(a) ❼  
Out[32]: 120
```

```
In [33]: a.__sizeof__() ❸  
Out[33]: 112
```

- ❶ Атрибут.
- ❷ Метод (с агрегирование данных).
- ❸ Метод (без агрегирования данных).
- ❹ Применение оператора + (сложение).
- ❺ Применение оператора \* (умножение).
- ❻ Применение функции `sum()` из стандартной библиотеки Python.
- ❼ Применение универсальной функции `np.sum()` из библиотеки NumPy.
- ❽ Вызов специального метода `__sizeof__()` для определения объема занимаемой объектом памяти.

## DataFrame

Последним мы рассмотрим объект `DataFrame` библиотеки `pandas`, который функционально напоминает объект `ndarray`. Для начала создадим объект `DataFrame` на основе объекта `ndarray`.

```
In [34]: import pandas as pd ❶
```

```
In [35]: df = pd.DataFrame(a, columns=list('abcd')) ❷
```

```
In [36]: type(df) ❸  
Out[36]: pandas.core.frame.DataFrame
```

- ❶ Импорт библиотеки `pandas`.
- ❷ Новый экземпляр `df`.
- ❸ Тип объекта.

Теперь познакомимся с его атрибутами и методами, а также операциями, которые он поддерживает.

```
In [37]: df.columns ❶  
Out[37]: Index(['a', 'b', 'c', 'd'], dtype='object')  
  
In [38]: df.sum() ❷
```

```
Out[38]: a    24
         b    28
         c    32
         d    36
         dtype: int64
```

```
In [39]: df.cumsum() ❸
```

```
Out[39]:
```

	a	b	c	d
0	0	0	1	2
1	4	6	8	10
2	12	15	18	21
3	24	28	32	36

```
In [40]: df + df ❹
```

```
Out[40]:
```

	a	b	c	d
0	0	0	2	4
1	8	10	12	14
2	16	18	20	22
3	24	26	28	30

```
In [41]: 2 * df ❺
```

```
Out[41]:
```

	a	b	c	d
0	0	0	2	4
1	8	10	12	14
2	16	18	20	22
3	24	26	28	30

```
In [42]: np.sum(df) ❻
```

```
Out[42]: a    24
         b    28
         c    32
         d    36
         dtype: int64
```

```
In [43]: df.__sizeof__() ❼
```

```
Out[43]: 208
```

- ❶ Атрибут.
- ❷ Метод (с агрегирование данных).
- ❸ Метод (без агрегирования данных).
- ❹ Применение оператора + (сложение).

- ⑤ Применение оператора `*` (умножение).
- ⑥ Применение универсальной функции `np.sum()` из библиотеки NumPy.
- ⑦ Вызов специального метода `__sizeof__()` для определения объема занимаемой объектом памяти.

## Основные операции с классами Python

В этом разделе описан синтаксис применения классов в Python. Мы рассмотрим создание пользовательских классов, моделирующих объекты, которые невозможно смоделировать с помощью встроенных типов Python. В качестве примера мы выберем модель *финансового инструмента*.

В Python для создания нового класса достаточно двух строчек кода.

```
In [44]: class FinancialInstrument(object): ❶  
        pass ❷
```

```
In [45]: fi = FinancialInstrument() ❸
```

```
In [46]: type(fi) ❹
```

```
Out[46]: __main__.FinancialInstrument
```

```
In [47]: fi ❺
```

```
Out[47]: <__main__.FinancialInstrument at 0x116767278>
```

```
In [48]: fi.__str__() ❻
```

```
Out[48]: '<__main__.FinancialInstrument object at 0x116767278>'
```

```
In [49]: fi.price = 100 ❼
```

```
In [50]: fi.price ❼
```

```
Out[50]: 100
```

- ❶ Определение класса<sup>2</sup>.
- ❷ Код класса; в данном случае это просто ключевое слово `pass`.
- ❸ Новый экземпляр класса с именем `fi`.

---

<sup>2</sup> Имена классов рекомендуется задавать в “горбатом” регистре (CamelCase). Впрочем, если риск неоднозначности отсутствует, то можно использовать нижний регистр или даже “змеиный” регистр (т.е. `financial_instrument`).

- ④ Тип объекта.
- ⑤ Каждый объект Python обладает рядом “специальных” атрибутов и методов (наследуемых от класса `object`). В данном случае вызывается специальный метод, возвращающий строковое представление объекта.
- ⑥ Атрибуты данных, в отличие от специальных атрибутов, можно создавать на лету.

Отдельного внимания заслуживает специальный метод `__init__()`, вызываемый при создании каждого экземпляра объекта. В качестве аргументов ему передается сам объект (представленный ключевым словом `self`) и произвольное число других параметров.

```
In [51]: class FinancialInstrument(object):  
         author = 'Ив Хилпиш' ①  
         def __init__(self, symbol, price): ②  
             self.symbol = symbol ③  
             self.price = price ③  
  
In [52]: FinancialInstrument.author ①  
Out[52]: 'Ив Хилпиш'  
  
In [53]: aapl = FinancialInstrument('AAPL', 100) ④  
  
In [54]: aapl.symbol ⑤  
Out[54]: 'AAPL'  
  
In [55]: aapl.author ⑥  
Out[55]: 'Ив Хилпиш'  
  
In [56]: aapl.price = 105 ⑦  
  
In [57]: aapl.price ⑦  
Out[57]: 105
```

- ① Определение атрибута класса (наследуется каждым экземпляром).
- ② Специальный метод `__init__()`, вызываемый при создании объекта.
- ③ Определение атрибутов экземпляра (разные у каждого объекта).
- ④ Новый экземпляр класса с именем `aapl`.
- ⑤ Получение атрибута экземпляра.

- ❹ Получение атрибута класса.
- ❺ Изменение атрибута экземпляра.

Стоимость финансовых инструментов постоянно меняется, а вот их тикеры обычно остаются неизменными. Чтобы продемонстрировать, как работает механизм инкапсуляции, создадим определение класса с двумя новыми методами: `get_price()` и `set_price()`. Теперь код класса наследуется от предыдущего определения, а не от класса `object`.

```
In [58]: class FinancialInstrument(FinancialInstrument):❶
        def get_price(self):❷
            return self.price❷
        def set_price(self, price):❸
            self.price = price❹
```

```
In [59]: fi = FinancialInstrument('AAPL', 100)❺
```

```
In [60]: fi.get_price()❻
Out[60]: 100
```

```
In [61]: fi.set_price(105)❼
```

```
In [62]: fi.get_price()❻
Out[62]: 105
```

```
In [63]: fi.price❽
Out[63]: 105
```

- ❶ Определение класса наследуется от предыдущей его версии.
- ❷ Определение метода `get_price()`.
- ❸ Определение метода `set_price()`...
- ❹ ...включающее обновление атрибута экземпляра, передаваемого в качестве параметра.
- ❺ Новый экземпляр `fi`, основанный на обновленном классе.
- ❻ Вызов метода `get_price()` для получения атрибута экземпляра.
- ❼ Обновление атрибута экземпляра с помощью метода `set_price()`.
- ❽ Прямой доступ к атрибуту экземпляра.

Инкапсуляция позволяет скрыть данные от пользователей класса. Частично это реализуется с помощью *методов доступа*, но они не препятствуют прямому обращению к атрибутам экземпляра. Здесь на помощь приходят *закрытые* атрибуты экземпляра, которые помечаются двумя начальными символами подчеркивания.

```
In [64]: class FinancialInstrument(object):
        def __init__(self, symbol, price):
            self.symbol = symbol
            self.__price = price ❶
        def get_price(self):
            return self.__price
        def set_price(self, price):
            self.__price = price
```

```
In [65]: fi = FinancialInstrument('AAPL', 100)
```

```
In [66]: fi.get_price() ❷
Out[66]: 100
```

```
In [67]: fi.__price ❸
```

```
-----
AttributeError      Traceback (most recent call last)
<ipython-input-67-bd62f6cadb79> in <module>
----> 1 fi.__price ❸
```

```
AttributeError: 'FinancialInstrument' object has no
attribute '__price'
```

```
In [68]: fi._FinancialInstrument__price ❹
Out[68]: 100
```

```
In [69]: fi._FinancialInstrument__price = 105 ❺
```

```
In [70]: fi.set_price(100) ❻
```

- ❶ Стоимость финансового инструмента хранится в закрытом атрибуте экземпляра.
- ❷ Метод `get_price()` возвращает значение закрытого атрибута.
- ❸ Попытка непосредственно обратиться к закрытому атрибуту приводит к ошибке.

- ④ Но если предварить имя атрибута именем класса с одиночным символом подчеркивания, то прямой доступ все еще можно получить.
- ⑤ Возврат цены к исходному значению.



### Инкапсуляция в Python

Несмотря на то что инкапсуляцию в Python можно реализовать с помощью закрытых атрибутов экземпляра и соответствующих методов доступа, полностью скрыть данные класса не представляется возможным. Это в большей степени инженерный принцип, заложенный в основу языка, а не техническая особенность классов в Python.

Рассмотрим еще один класс, моделирующий портфельную позицию финансового инструмента. На примере двух классов легко продемонстрировать механизм *агрегирования*. Экземпляр класса `PortfolioPosition` получает экземпляр класса `FinancialInstrument` в качестве своего атрибута. Добавив соответствующий атрибут экземпляра, скажем, `position_size`, можно будет вычислить стоимость позиции.

```
In [71]: class PortfolioPosition(object):
        def __init__(self, financial_instrument, position_size):
            self.position = financial_instrument ①
            self.__position_size = position_size ②
        def get_position_size(self):
            return self.__position_size
        def update_position_size(self, position_size):
            self.__position_size = position_size
        def get_position_value(self):
            return self.__position_size * \
                self.position.get_price() ③
```

```
In [72]: pp = PortfolioPosition(fi, 10)
```

```
In [73]: pp.get_position_size()
```

```
Out[73]: 10
```

```
In [74]: pp.get_position_value() ③
```

```
Out[74]: 1000
```

```
In [75]: pp.position.get_price() ④
```

```
Out[75]: 100
```

```
In [76]: pp.position.set_price(105) ⑤
```



```
In [77]: pp.get_position_value() ⑥  
Out[77]: 1050
```

- ① Атрибут экземпляра, представленный экземпляром класса `FinancialInstrument`.
- ② Замкнутый атрибут экземпляра класса `PortfolioPosition`.
- ③ Вычисление стоимости позиции на основе известных атрибутов.
- ④ Если атрибутом экземпляра является объект, то его методы можно вызывать напрямую.
- ⑤ Обновление стоимости финансового инструмента.
- ⑥ Вычисление новой стоимости портфельной позиции на основе обновленной стоимости финансового инструмента.

## Модель данных Python

В примерах предыдущего раздела демонстрировались определенные аспекты *модели данных* Python (<https://docs.python.org/3/reference/datamodel.html>). Она позволяет создавать собственные классы, согласованным образом взаимодействующие с базовыми конструкциями языка. Среди прочего модель поддерживает следующие возможности:

- циклические операции;
- обработка коллекций;
- доступ к атрибутам;
- перегрузка операторов;
- вызов методов и функций;
- создание и удаление объектов;
- строковое представление объектов (например, для вывода информации на экран);
- управление контекстом (например, в блоках `with`).

Модель данных Python настолько важна, что необходимо рассмотреть ее особенности на конкретных примерах. Для этого мы создадим класс, описывающий трехэлементный вектор. Сначала реализуем специальный метод `__init__()`.

```
In [78]: class Vector(object):
        def __init__(self, x=0, y=0, z=0): ❶
            self.x = x ❶
            self.y = y ❶
            self.z = z ❶
```

```
In [79]: v = Vector(1, 2, 3) ❷
```

```
In [80]: v ❸
Out[80]: <__main__.Vector at 0x1167789e8>
```

- ❶ Три предварительно заданных атрибута экземпляра (координаты в трехмерном пространстве).
- ❷ Новый экземпляр класса с именем `v`.
- ❸ Вывод строкового представления экземпляра, заданного по умолчанию.

Специальный метод `__repr__()` позволяет задать пользовательское строковое представление класса.

```
In [81]: class Vector(Vector):
        def __repr__(self):
            return 'Vector(%r, %r, %r)' %
                (self.x, self.y, self.z)
```

```
In [82]: v = Vector(1, 2, 3)
```

```
In [83]: v ❶
Out[83]: Vector(1, 2, 3)
```

```
In [84]: print(v) ❶
Vector(1, 2, 3)
```

- ❶ Новый способ строкового представления класса.

В Python есть стандартные функции `abs()` и `bool()`, поведение которых применительно к классу `Vector` можно настроить с помощью специальных методов `__abs__()` и `__bool__()`.

```
In [85]: class Vector(Vector):
        def __abs__(self):
            return (self.x ** 2 + self.y ** 2 +
                    self.z ** 2) ** 0.5 ❶
        def __bool__(self):
            return bool(abs(self))
```

```
In [86]: v = Vector(1, 2, -1) ❷
```

```
In [87]: abs(v)
```

```
Out[87]: 2.449489742783178
```

```
In [88]: bool(v)
```

```
Out[88]: True
```

```
In [89]: v = Vector() ❸
```

```
In [90]: v ❸
```

```
Out[90]: Vector(0, 0, 0)
```

```
In [91]: abs(v)
```

```
Out[91]: 0.0
```

```
In [92]: bool(v)
```

```
Out[92]: False
```

- ❶ Возвращает евклидову норму по заданным значениям трех атрибутов.
- ❷ Новый объект `Vector` с ненулевыми значениями атрибутов.
- ❸ Новый объект `Vector` с нулевыми значениями атрибутов.

Как уже было неоднократно показано, операторы `+` и `*` применяются к большинству объектов Python, а их поведение определяется специальными методами `__add__()` и `__mul__()`.

```
In [93]: class Vector(Vector):  
    def __add__(self, other):  
        x = self.x + other.x  
        y = self.y + other.y  
        z = self.z + other.z  
        return Vector(x, y, z) ❶  
  
    def __mul__(self, scalar):  
        return Vector(self.x * scalar,  
                        self.y * scalar,  
                        self.z * scalar) ❶
```

```
In [94]: v = Vector(1, 2, 3)
```

```
In [95]: v + Vector(2, 3, 4)
```

```
Out[95]: Vector(3, 5, 7)
```

```
In [96]: v * 2
```

```
Out[96]: Vector(2, 4, 6)
```

- ❶ В данном случае каждый специальный метод возвращает объект того же типа.

Еще одна стандартная функция Python — `len()` — возвращает длину объекта, выраженную в виде количества содержащихся в нем элементов. При получении пользовательского объекта функция вызывает специальный метод `__len__()`. Существует также специальный метод `__getitem__()`, который делает возможной индексацию через квадратные скобки.

```
In [97]: class Vector(Vector):
        def __len__(self):
            return 3 ❶
        def __getitem__(self, i):
            if i in [0, -3]: return self.x
            elif i in [1, -2]: return self.y
            elif i in [2, -1]: return self.z
            else: raise IndexError('Индекс за пределами
                                   диапазона.')
```

```
In [98]: v = Vector(1, 2, 3)
```

```
In [99]: len(v)
```

```
Out[99]: 3
```

```
In [100]: v[0]
```

```
Out[100]: 1
```

```
In [101]: v[-2]
```

```
Out[101]: 2
```

```
In [102]: v[3]
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-102-f998c57dcc1e> in <module>
----> 1 v[3]

<ipython-input-97-b0ca25eef7b3> in __getitem__(self, i)
      7     elif i in [1, -2]: return self.y
```

```

8     elif i in [2, -1]: return self.z
----> 9     else: raise IndexError('Индекс за пределами
                                диапазона.')

```

`IndexError: Индекс за пределами диапазона.`

- ❶ Все экземпляры класса `Vector` имеют длину 3.

Наконец, специальный метод `__iter__()` определяет поведение объекта в операциях циклического перебора его элементов. Объект, для которого определен такой метод, называется *итератором*. В частности, итераторами являются любые последовательности и контейнеры.

```

In [103]: class Vector(Vector):
           def __iter__(self):
               for i in range(len(self)):
                   yield self[i]

```

```

In [104]: v = Vector(1, 2, 3)

```

```

In [105]: for i in range(3): ❶
           print(v[i]) ❶
1
2
3

```

```

In [106]: for coordinate in v: ❷
           print(coordinate) ❷
1
2
3

```

- ❶ Неявный перебор через индексы (вызывается специальный метод `__getitem__()`).
- ❷ Явное итерирование через интерфейс класса (вызывается специальный метод `__iter__()`).



### Расширение объектной модели Python

Модель данных Python позволяет определять классы, неявно взаимодействующие со стандартными операторами, функциями и т.п. Такой подход делает Python невероятно гибким языком программирования, возможности которого легко расширяются за счет новых классов и типов объектов.

## Код класса Vector

Подытоживая все вышесказанное, запишем полное определение класса Vector.

```
In [107]: class Vector(object):
    def __init__(self, x=0, y=0, z=0):
        self.x = x
        self.y = y
        self.z = z

    def __repr__(self):
        return 'Vector(%r, %r, %r)' %
            (self.x, self.y, self.z)

    def __abs__(self):
        return (self.x ** 2 + self.y ** 2 +
            self.z ** 2) ** 0.5

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Vector(x, y, z)

    def __mul__(self, scalar):
        return Vector(self.x * scalar,
            self.y * scalar,
            self.z * scalar)

    def __len__(self):
        return 3

    def __getitem__(self, i):
        if i in [0, -3]: return self.x
        elif i in [1, -2]: return self.y
        elif i in [2, -1]: return self.z
        else: raise IndexError('Индекс за пределами
            диапазона.')

    def __iter__(self):
        for i in range(len(self)):
            yield self[i]
```

## Резюме

В этой главе был дан обзор принципов объектно-ориентированного программирования на Python. ООП — одна из ключевых парадигм программирования, применяемых в Python. Она позволяет не только разрабатывать сложные приложения, но и создавать собственные объекты, которые функционируют подобно стандартным объектам языка благодаря гибкой модели данных. ООП предоставляет в распоряжение разработчиков мощные инструменты, незаменимые в решении многих финансовых задач. В качестве примера можно привести пакет оценки деривативов, который мы разработаем в части V. В этом случае ООП оказывается единственной эффективной парадигмой, позволяющей справиться с возникающими трудностями и требованиями к абстрагированию финансовых данных.

## Дополнительные ресурсы

Следующие ресурсы представляют интерес для изучения ООП в целом и объектно-ориентированного программирования на Python в частности:

- конспект лекций по объектно-ориентированному программированию (<https://bit.ly/2qLJU0S>).
- объектно-ориентированное программирование на Python (<https://bit.ly/2DKGZhB>).

Великолепным источником информации по объектно-ориентированному программированию на Python послужит следующая книга:

- Ramalho, Luciano. *Fluent Python* (2016, O'Reilly).

---

# Обработка и анализ финансовых данных

Эта часть посвящена инструментам и пакетам, предназначенным для работы с финансовыми данными. Многие из рассматриваемых здесь тем (например, визуализация данных) и библиотек (таких, как Scikit-learn) являются фундаментальными с точки зрения применения науки о данных в Python. Но мы сконцентрируемся на финансовых вычислениях.

Как и в части II, главы данной части посвящены конкретным темам и служат справочным руководством, к которому читатели смогут обращаться в случае необходимости.

- **Глава 7** посвящена построению статических и интерактивных диаграмм с помощью пакетов `matplotlib` и `plotly`.
- **Глава 8** посвящена обработке временных рядов с помощью библиотеки `pandas`.
- **Глава 9** посвящена выполнению операций ввода-вывода.
- **Глава 10** содержит рекомендации по повышению производительности кода Python.
- **Глава 11** посвящена математическому аппарату финансовых расчетов.
- **Глава 12** посвящена реализации стохастических методов в Python.
- **Глава 13** посвящена статистическим расчетам и методам машинного обучения.





---

# Визуализация данных

Используйте иллюстрацию. Она стоит тысячи слов.

*Артур Брисбен*

В этой главе рассматриваются средства визуализации данных, доступные в пакетах `matplotlib` и `plotly`.

Несмотря на наличие других мощных библиотек визуализации, пакет `matplotlib` ([www.matplotlib.org](http://www.matplotlib.org)) считается эталоном, предлагающим самые надежные и эффективные решения. С одной стороны, он позволяет легко строить стандартные графики; с другой стороны, он обеспечивает достаточную гибкость при настройке сложных диаграмм. Кроме того, он тесно интегрирован с библиотеками `NumPy` и `pandas` и поддерживает реализованные в них структуры данных.

К сожалению, пакет `matplotlib` способен генерировать диаграммы только в растровом формате (например, PNG или JPEG). В то же время существуют современные веб-технологии, такие как стандарт D3.js (<https://d3js.org>), которые позволяют создавать интерактивные встраиваемые графические объекты (с поддержкой масштабирования отдельных областей). В Python за построение такого рода диаграмм отвечает пакет `plotly` (<http://plot.ly>). Есть также небольшая вспомогательная библиотека `Cufflinks`, которая интегрирует пакет `plotly` с объектами `DataFrame` библиотеки `pandas` и поддерживает популярные финансовые диаграммы (в частности, “японские свечи”).

В главе рассматриваются следующие темы.

## *Статические двумерные графики*

В этом разделе мы познакомимся с библиотекой `matplotlib` и узнаем, как строить типичные графики.

## *Статические трехмерные диаграммы*

В этом разделе рассматриваются способы построения трехмерных диаграмм, применяемых в финансовых приложениях.

## Интерактивные двухмерные диаграммы

В этом разделе вы узнаете, как с помощью пакетов `plotly` и `Cufflinks` создавать интерактивные двухмерные диаграммы. Также будет рассмотрен процесс построения типичных финансовых диаграмм, применяемых, к примеру, в техническом анализе.

Данная глава не является исчерпывающим руководством по визуализации в Python, а также пакетам `matplotlib` и `plotly`. Тем не менее в ней содержится множество примеров применения этих пакетов в задачах финансового анализа. В последующих главах будут рассмотрены и другие примеры. В частности, глава 8 будет посвящена визуализации финансовых временных рядов средствами библиотеки `pandas`.

## Статические двухмерные графики

Прежде чем формировать выборку данных и строить графики, необходимо импортировать соответствующие модули и задать ряд базовых настроек.

```
In [1]: import matplotlib as mpl ❶
```

```
In [2]: mpl.__version__ ❷  
Out[2]: '3.0.0'
```

```
In [3]: import matplotlib.pyplot as plt ❸
```

```
In [4]: plt.style.use('seaborn') ❹
```

```
In [5]: mpl.rcParams['font.family'] = 'serif' ❺
```

```
In [6]: %matplotlib inline
```

- ❶ Импорт пакета `matplotlib`, представленного аббревиатурой `mpl`.
- ❷ Версия пакета `matplotlib`.
- ❸ Импорт основного модуля построения диаграмм, представленного аббревиатурой `plt`.
- ❹ Выбор стиля `seaborn` (<http://bit.ly/2KaPFhs>)
- ❺ Выбор гарнитуры `serif` для всех графиков.

## Одномерные наборы данных

Основная функция построения диаграмм — `plt.plot()`. Ей необходимо передать два набора значений.

*Координаты  $x$*

Список или массив, содержащий координаты  $x$  (значения по оси абсцисс).

*Координаты  $y$*

Список или массив, содержащий координаты  $y$  (значения по оси ординат).

Разумеется, количество значений в обоих наборах должно совпадать. Рассмотрим следующий пример, результат выполнения которого представлен на рис. 7.1.

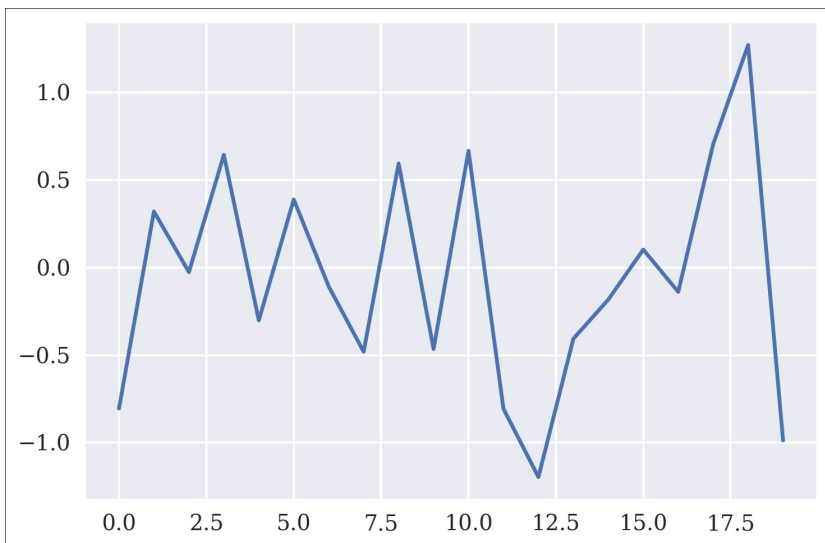


Рис. 7.1. График для заданных значений  $x$  и  $y$

```
In [7]: import numpy as np
```

```
In [8]: np.random.seed(1000) ❶
```

```
In [9]: y = np.random.standard_normal(20) ❷
```

```
In [10]: x = np.arange(len(y)) ❸  
         plt.plot(x, y); ❹
```

- ❶ Фиксируем затравочное значение генератора псевдослучайных чисел, чтобы пример можно было легко воспроизвести.
- ❷ Получение набора псевдослучайных чисел (значений  $y$ ).
- ❸ Получение диапазона целых чисел (значений  $x$ ).
- ❹ Вызов функции `plt.plot()` и передача ей объектов  $x$  и  $y$ .

Функция `plt.plot()` автоматически распознает объекты `ndarray`, поэтому в данном случае значения  $x$  избыточны. Если передаются только значения  $y$ , функция берет соответствующие индексы в качестве значений  $x$ . Это означает, что следующая строка кода сгенерирует точно такой же график (рис. 7.2).

```
In [11]: plt.plot(y);
```

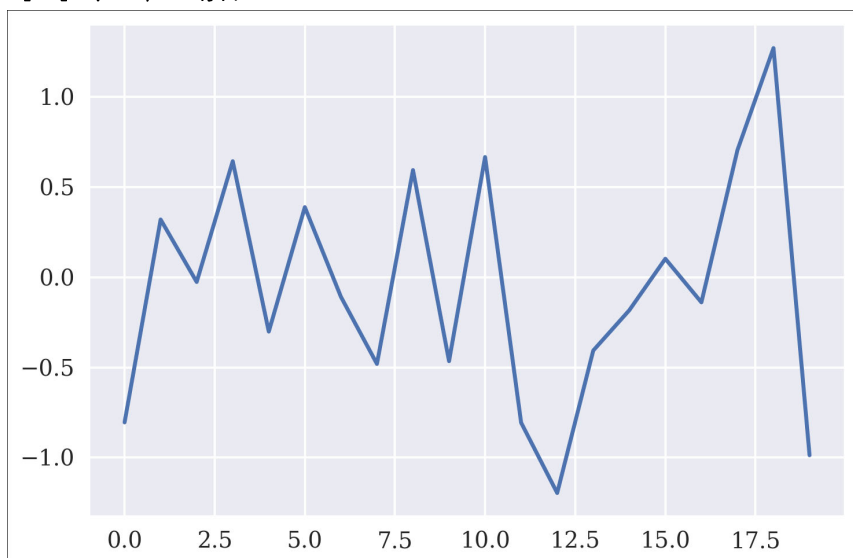


Рис. 7.2. График для заданного массива `ndarray`



### Массивы NumPy и пакет matplotlib

Функциям `matplotlib` можно передавать объекты `ndarray` библиотеки NumPy, что позволяет существенно сократить код построения диаграмм. Но следите за тем, чтобы передаваемые массивы не были слишком большими и/или сложными.

Поскольку большинство методов объекта `ndarray` в свою очередь возвращает массив `ndarray`, при выполнении функции `plt.plot()` можно вызвать определенный метод (или даже цепочку методов). Например, при вызове метода `cumsum()` будет построен график изменения накопительной суммы (рис. 7.3).

```
In [12]: plt.plot(y.cumsum());
```

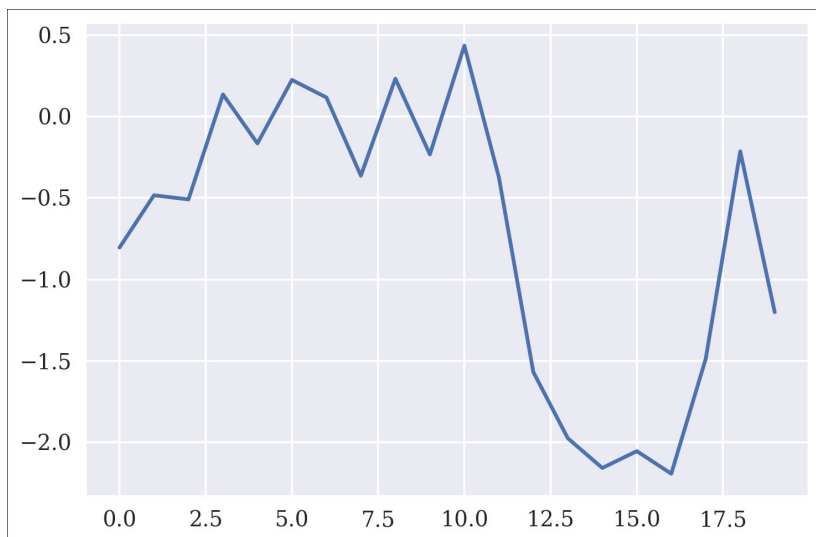


Рис. 7.3. График для заданного массива `ndarray` с вызовом метода `cumsum()`

Заданный по умолчанию стиль диаграммы далеко не всегда соответствует требованиям публикации. Чаще всего приходится изменять шрифты (например, для обеспечения совместимости со шрифтами LaTeX), добавлять подписи к осям, отображать координатную сетку и т.п. Проще всего использовать готовые стили диаграмм. Кроме того, в пакете `matplotlib` имеется большое число функций, предназначенных для настройки диаграмм. Некоторые из них просты и понятны, а некоторые требуют более детального изучения. Легче всего разобраться с функциями, которые отвечают за настройку координатной сетки и осей (рис. 7.4).

```
In [13]: plt.plot(y.cumsum())  
         plt.grid(False) ❶  
         plt.axis('equal'); ❷
```

- ❶ Скрывает координатную сетку.
- ❷ Выравнивает масштаб обеих осей.

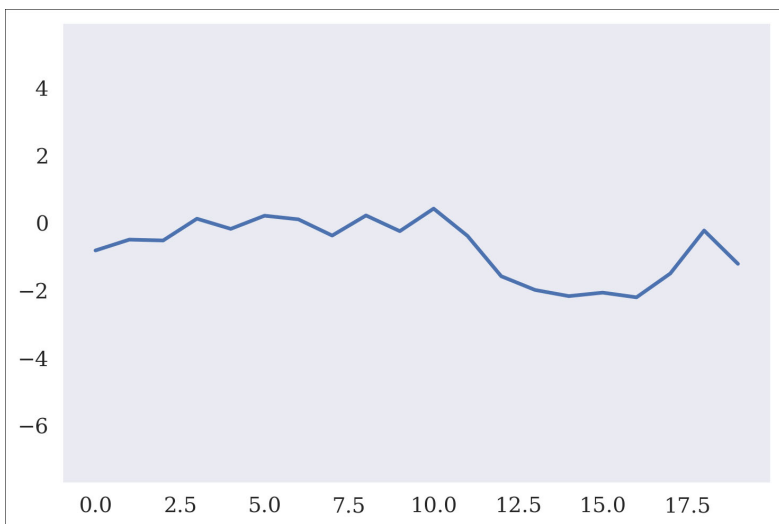


Рис. 7.4. График без координатной сетки

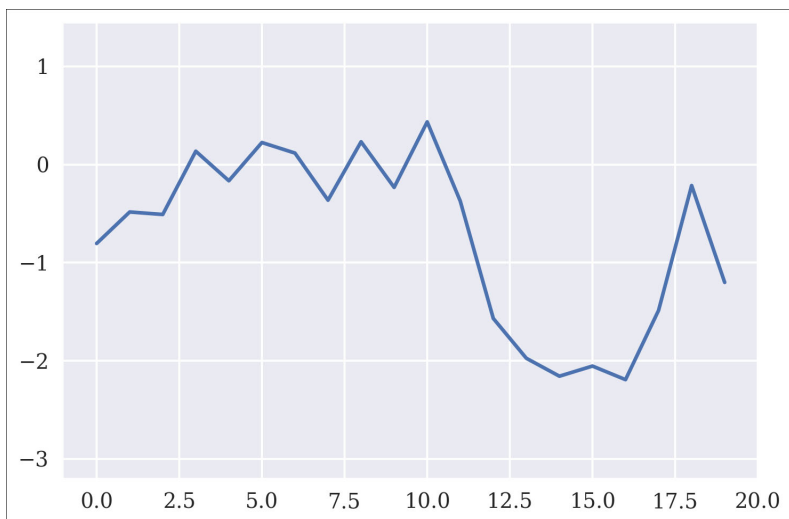
Различные параметры функции `plt.axis()` перечислены в табл. 7.1. В основном это должны быть объекты `str`.

Таблица 7.1. Параметры функции `plt.axis()`

Параметр	Описание
—	Возвращает текущие предельные значения осей
<code>off</code>	Скрывает линии осей и подписи к ним
<code>equal</code>	Выравнивает масштаб осей
<code>scaled</code>	Выравнивает масштаб за счет изменения размерностей
<code>tight</code>	Приводит к отображению всех данных (за счет сжатия осей)
<code>image</code>	Приводит к отображению всех данных (предельные значения определяются данными)
<code>[xmin, xmax, ymin, ymax]</code>	Устанавливает предельные значения на осях

Можно также непосредственно задать минимальные и максимальные значения каждой оси, воспользовавшись методами `plt.xlim()` и `plt.ylim()`. Рассмотрим пример (рис. 7.5).

```
In [14]: plt.plot(y.cumsum())
          plt.xlim(-1, 20)
          plt.ylim(np.min(y.cumsum()) - 1,
                  np.max(y.cumsum()) + 1);
```



*Рис. 7.5. График с измененными пределами осей*

Для наглядности график обычно снабжают всевозможными надписями, например заголовком и подписями осей  $x$  и  $y$ . Они добавляются с помощью функций `plt.title()`, `plt.xlabel()` и `plt.ylabel()`. По умолчанию функция `plot()` строит непрерывный график даже для дискретных данных. Чтобы получить график с маркерами, необходимо выбрать другой стиль. В следующем примере выводятся красные маркеры, а соединительные линии имеют синий цвет и толщину 1,5 пункта (рис. 7.6).

```
In [15]: plt.figure(figsize=(10, 6)) ❶
          plt.plot(y.cumsum(), 'b', lw=1.5) ❷
          plt.plot(y.cumsum(), 'ro') ❸
          plt.xlabel('Индекс') ❹
          plt.ylabel('Значение') ❺
          plt.title('Простой график'); ❻
```

- ❶ Увеличение размера рисунка.
- ❷ Рисование графика синей линией толщиной 1,5 пункта.
- ❸ Нанесение красных точек данных.
- ❹ Добавление подписи к оси  $X$ .
- ❺ Добавление подписи к оси  $Y$ .
- ❻ Добавление заголовка графика.



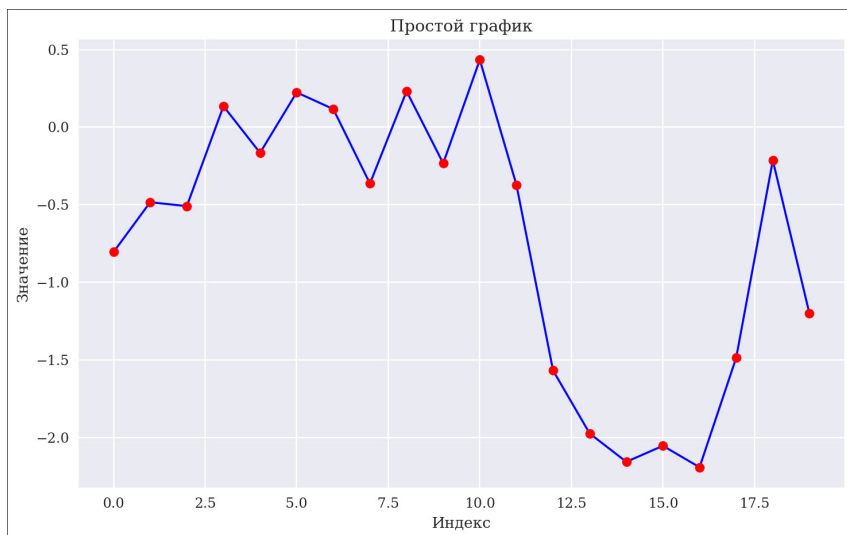


Рис. 7.6. График с подписями

По умолчанию функция `plt.plot()` поддерживает следующие цветовые обозначения (табл. 7.2).

Таблица 7.2. Стандартные цветовые обозначения

Символ	Цвет
b	Синий
g	Зеленый
r	Красный
c	Голубой
m	Пурпурный
y	Желтый
k	Черный
w	Белый

Для обозначения стиля линий и маркеров в функции `plt.plot()` применяются символы, приведенные в табл. 7.3.

Обозначение стиля допускается комбинировать с обозначением цвета. Это позволяет различать разные наборы данных на одном графике. Учтите, что стиль линии также выносится в легенду.

Таблица 7.3. Стандартные обозначения стилей

Символ	Стиль
-	Сплошная линия
--	Штриховая линия
-.	Штрих-пунктирная линия
:	Пунктирная линия
.	Маркер в виде точки
,	Маркер в виде пикселя
o	Маркер в виде кружка
v	Маркер в виде треугольника с вершиной вниз
^	Маркер в виде треугольника с вершиной вверх
<	Маркер в виде треугольника с вершиной влево
>	Маркер в виде треугольника с вершиной вправо
1	Маркер в виде трехлучевой звезды с вершиной вниз
2	Маркер в виде трехлучевой звезды с вершиной вверх
3	Маркер в виде трехлучевой звезды с вершиной влево
4	Маркер в виде трехлучевой звезды с вершиной вправо
s	Маркер в виде квадрата
p	Маркер в виде пятиугольника
*	Маркер в виде звездочки
h	Маркер в виде шестиугольника, тип 1
H	Маркер в виде шестиугольника, тип 2
+	Маркер в виде знака "плюс"
x	Маркер в виде символа X
D	Маркер в виде ромба
d	Маркер в виде узкого ромба
	Маркер в виде вертикальной линии
_	Маркер в виде горизонтальной линии

## Двухмерные наборы данных

Построение графика одномерного набора данных — это скорее исключение, чем правило. В большинстве случаев наборы данных включают несколько обособленных поднаборов. Для их обработки в пакете `matplotlib`

применяются те же функции, что и в “одномерном” случае. Тем не менее при работе с двухмерными данными следует учитывать определенные ограничения. Например, поднаборы могут иметь совершенно разный масштаб, что не позволит наглядно представить их на общем графике. Кроме того, может понадобиться визуализировать каждый из наборов по-своему: один — в виде линейного графика, другой — в виде гистограммы.

Приведенный ниже фрагмент кода формирует двухмерный набор данных в виде массива `ndarray` размером  $20 \times 2$ , содержащий псевдослучайные числа с нормальным распределением. К массиву применяется метод `cumsum()` для вычисления накопительной суммы значений по оси 0 (т.е. по первому измерению).

In [16]: `y = np.random.standard_normal((20, 2)).cumsum(axis=0)`

Часто такой массив напрямую передается функции `plt.plot()`, которая автоматически распознает все поднаборы (в нашем случае ось 1, т.е. второе измерение). Результат представлен на рис. 7.7.

In [17]: `plt.figure(figsize=(10, 6))`  
`plt.plot(y, lw=1.5)`  
`plt.plot(y, 'ro')`  
`plt.xlabel('Индекс')`  
`plt.ylabel('Значение')`  
`plt.title('Простой график');`

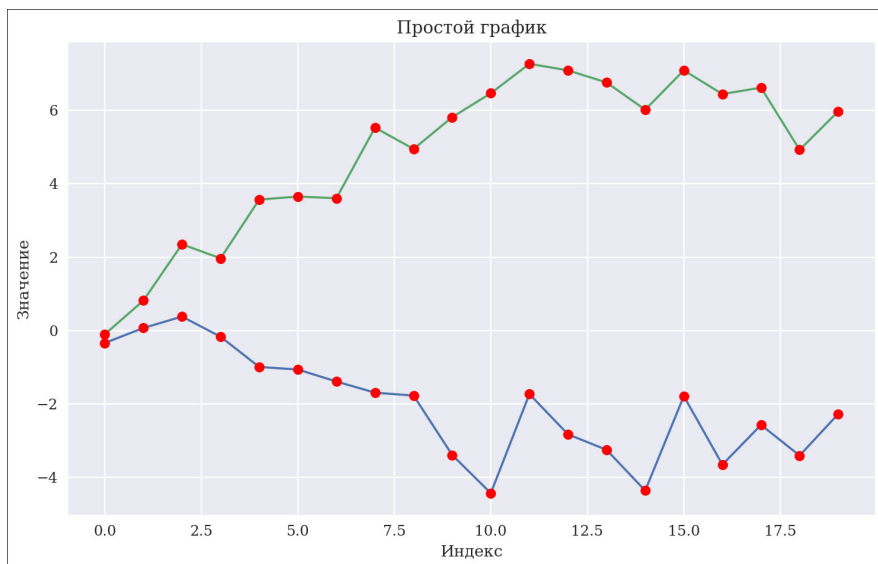


Рис. 7.7. График двухмерного набора данных

В подобных случаях имеет смысл повысить наглядность диаграммы, добавив к ней легенду с текстовым описанием каждого набора данных. Функция `plt.legend()` поддерживает различные параметры, определяющие местоположение легенды. В частности, значение `0` предписывает располагать легенду в наиболее удобном месте, т.е. так, чтобы она как можно меньше перекрывала визуализируемые данные.

На рис. 7.8 показан график двух наборов данных, снабженный легендой. На этот раз объект `ndarray` не передается напрямую — исходные данные представлены здесь двумя отдельными поднаборами (`y[:, 0]` и `y[:, 1]`), что позволяет назначить им разные подписи в легенде.

```
In [18]: plt.figure(figsize=(10, 6))
plt.plot(y[:, 0], lw=1.5, label='1-й') ❶
plt.plot(y[:, 1], lw=1.5, label='2-й') ❶
plt.plot(y, 'ro')
plt.legend(loc=0) ❷
plt.xlabel('Индекс')
plt.ylabel('Значение')
plt.title('Простой график');
```

- ❶ Подписи к поднаборам данных.
- ❷ Указывает размещать легенду в наиболее удобном месте.

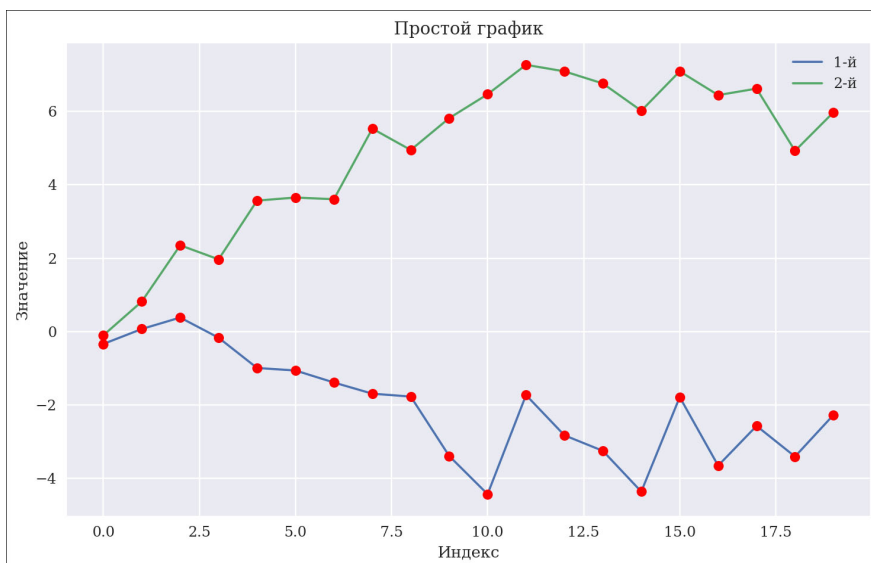


Рис. 7.8. График с легендой

Расположение легенды на графике можно задавать параметрами метода `plt.legend()`, описанными в табл. 7.4.

Таблица 7.4. Параметры метода `plt.legend()`

Расположение	Описание
По умолчанию	Справа вверху
0	В самом удобном месте
1	Справа вверху
2	Слева вверху
3	Слева внизу
4	Справа внизу
5	Справа
6	Слева по центру
7	Справа по центру
8	Внизу по центру
9	Вверху по центру
10	По центру

Множественные наборы данных, заданные в одном масштабе, например прогнозы различных моделей рисков, можно отображать вдоль одной оси Y. Но зачастую визуализируемые поднаборы имеют разный масштаб, и попытка построить такие графики вдоль общей оси Y приведет к потере наглядности. Чтобы проиллюстрировать сказанное, масштабируем первый из наборов с коэффициентом 100 и построим тот же самый график (рис. 7.9).

```
In [19]: y[:, 0] = y[:, 0] * 100 ❶
```

```
In [20]: plt.figure(figsize=(10, 6))
plt.plot(y[:, 0], lw=1.5, label='1-й')
plt.plot(y[:, 1], lw=1.5, label='2-й')
plt.plot(y, 'ro')
plt.legend(loc=0)
plt.xlabel('Индекс')
plt.ylabel('Значение')
plt.title('Простой график');
```

❶ Изменение масштаба исходного набора данных.

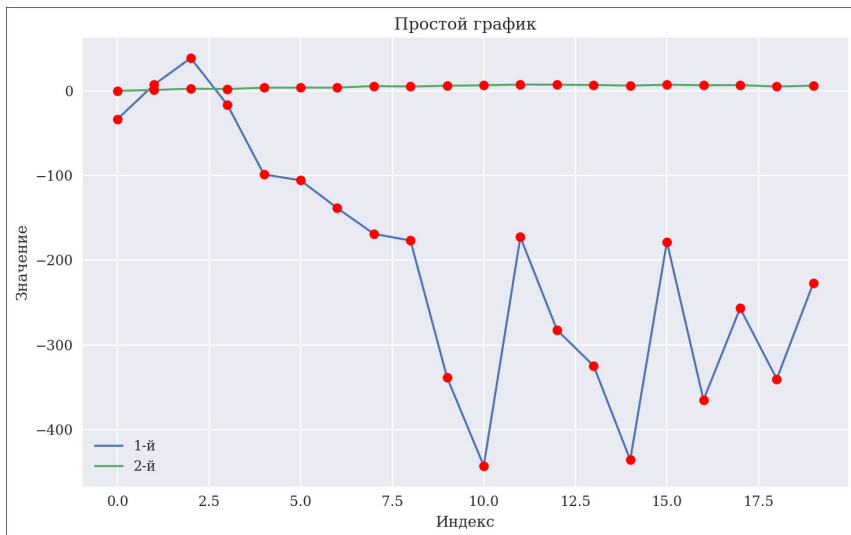


Рис. 7.9. Построение двух разных графиков в разном масштабе

Как видим, первый набор данных по-прежнему визуализируется очень наглядно, тогда как второй график фактически превратился в прямую линию вследствие резкого изменения масштаба оси Y, что делает его практически бесполезным для анализа. Существуют два подхода к решению такого рода проблем графическим путем (т.е. без корректировки исходных данных):

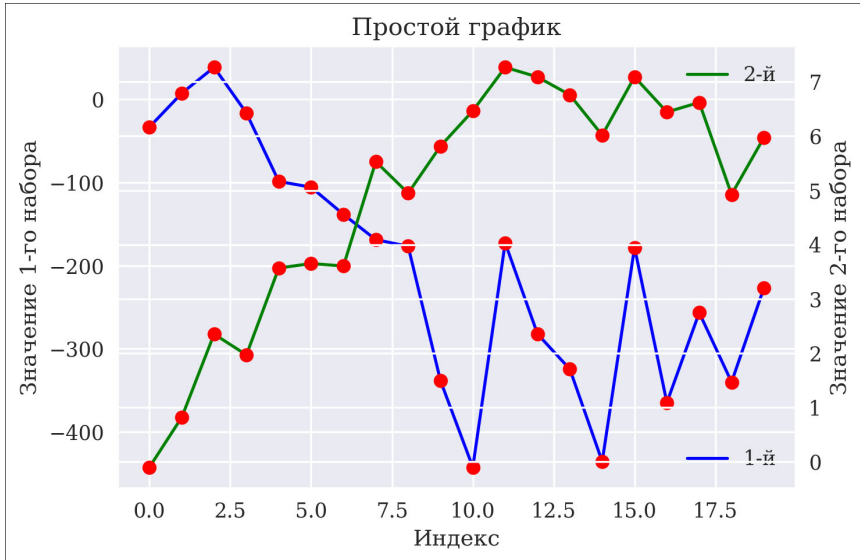
- использование двух осей Y (слева/справа);
- использование двух графических областей (вверху/внизу или слева/справа).

В следующем примере к области построения добавляется вторая ось Y (рис. 7.10). Левая ось соответствует первому набору значений, а правая ось — второму. Это требует отображения двух подписей оси.

```
In [21]: fig, ax1 = plt.subplots() ❶
plt.plot(y[:, 0], 'b', lw=1.5, label='1-й')
plt.plot(y[:, 0], 'ro')
plt.legend(loc=4)
plt.xlabel('Индекс')
plt.ylabel('Значение 1-го набора')
plt.title('Простой график')
ax2 = ax1.twinx() ❷
plt.plot(y[:, 1], 'g', lw=1.5, label='2-й')
```

```
plt.plot(y[:, 1], 'ro')
plt.legend(loc=0)
plt.ylabel('Значение 2-го набора');
```

- ❶ Определение объектов области построения и базовых осей.
- ❷ Создание объекта второй оси Y для общей оси X.



*Рис. 7.10. Графики двух поднаборов данных, отображаемые в разных координатах y*

Ключевую роль здесь играют следующие две строки кода.

```
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
```

Функция `plt.subplots()` позволяет получить доступ к различным объектам диаграммы (область построения, графики и т.п.). Это позволяет, например, сгенерировать второй график, который будет отложен вдоль той же оси X, что и первый. Как показано на рис. 7.10, оба графика фактически накладываются друг на друга.

Теперь рассмотрим случай с *независимыми* наборами данных. Это дает нам больше свободы в выборе визуального представления каждого из наборов (рис. 7.11).

```
In [22]: plt.figure(figsize=(10, 6))
plt.subplot(211) ❶
plt.plot(y[:, 0], lw=1.5, label='1-й')
plt.plot(y[:, 0], 'ro')
plt.legend(loc=0)
plt.ylabel('Значение')
plt.title('Простой график')
plt.subplot(212) ❷
plt.plot(y[:, 1], 'g', lw=1.5, label='2-й')
plt.plot(y[:, 1], 'ro')
plt.legend(loc=0)
plt.xlabel('Индекс')
plt.ylabel('Значение');
```

- ❶ Определение верхней области построения для первого набора.
- ❷ Определение нижней области построения для второго набора.

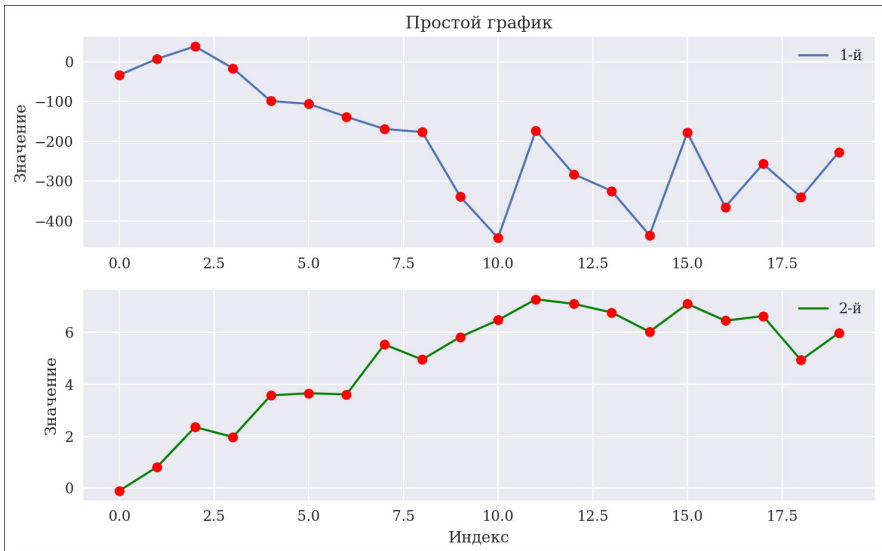


Рис. 7.11. Два графика в пределах общей области построения

Включение нескольких областей построения в единый контейнер `matplotlib` требует специальной системы координат. Для этого используется функция `plt.subplot()`, которая поддерживает три целочисленных аргумента: `numrows`, `numcols` и `fignum` (могут быть разделены запятыми либо записаны подряд). Параметр `numrows` определяет количество строк, параметр



`numcols` — количество столбцов, а параметр `fignum` — номер области построения, начиная с 1 и заканчивая значением `numrows * numcols`. В частности, графический контейнер, включающий девять областей одинакового размера (`fignum=1,2,3,...,9`), будет состоять из трех строк (`numrows=3`) и трех столбцов (`numcols=3`). Соответственно, правая нижняя область построения будет определяться “координатами” `plt.subplot(3, 3, 9)`.

Иногда визуализация такого рода данных выполняется с помощью диаграмм различного типа. При этом допускается комбинировать любые виды диаграмм, поддерживаемые в `matplotlib`<sup>1</sup>.

На рис. 7.12 показано, как в одном контейнере совмещаются график и гистограмма.

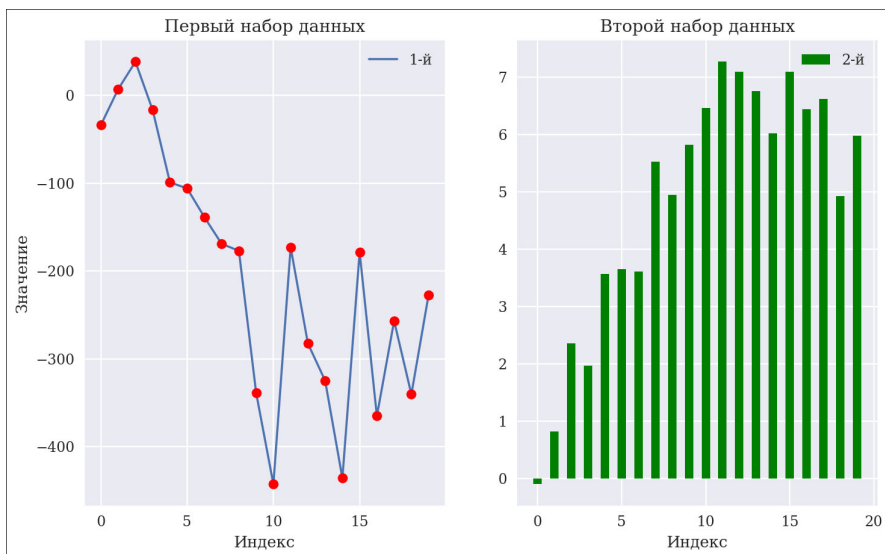


Рис. 7.12. Совмещение графика и гистограммы в одном контейнере

```
In [23]: plt.figure(figsize=(10, 6))
plt.subplot(121)
plt.plot(y[:, 0], lw=1.5, label='1-й')
plt.plot(y[:, 0], 'ro')
plt.legend(loc=0)
plt.xlabel('Индекс')
plt.ylabel('Значение')
plt.title('Первый набор данных')
```

<sup>1</sup> Всевозможные типы диаграмм `matplotlib` описаны на официальном сайте (<https://matplotlib.org/gallery.html>).

```
plt.subplot(122)
plt.bar(np.arange(len(y)), y[:, 1], width=0.5,
        color='g', label='2-й') ❶
plt.legend(loc=0)
plt.xlabel('Индекс')
plt.title('Второй набор данных');
```

❶ Создание гистограммы для поднабора данных.

## Другие типы диаграмм

В финансовом моделировании чаще всего применяются графики и точечные диаграммы, поскольку это лучший способ визуализации временных рядов. Детальнее с финансовыми временными рядами мы познакомимся в следующей главе, а пока поработаем с двухмерным набором случайных значений, рассмотрев альтернативные способы его визуализации, полезные с точки зрения финансового анализа.

Первый способ — это *диаграмма рассеяния* (scatter plot), в которой первый набор данных служит значениями  $x$  для второго набора (рис. 7.13). Такой способ визуализации может, к примеру, применяться для сопоставления уровней доходности двух финансовых временных рядов. Сгенерируем новый двухмерный набор данных.

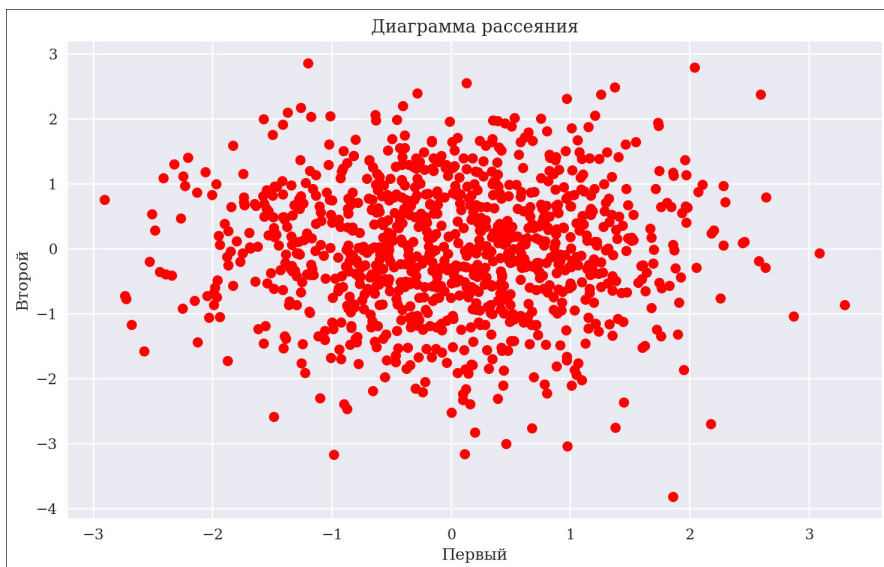


Рис. 7.13. Диаграмма рассеяния, созданная функцией `plt.plot()`

```
In [24]: y = np.random.standard_normal((1000, 2)) ❶
```

```
In [25]: plt.figure(figsize=(10, 6))  
plt.plot(y[:, 0], y[:, 1], 'ro') ❷  
plt.xlabel('Первый')  
plt.ylabel('Второй')  
plt.title('Диаграмма рассеяния');
```

- ❶ Создание большого набора случайных значений.
- ❷ Построение диаграммы рассеяния с помощью функции `plt.plot()`.

В пакете `matplotlib` имеется функция `plt.scatter()`, предназначенная для построения диаграмм рассеяния. Она работает так же, как и функция `plt.plot()`, но поддерживает ряд дополнительных параметров. На рис. 7.14 представлена та же самая диаграмма, что и на рис. 7.13, но на этот раз созданная функцией `plt.scatter()`.

```
In [26]: plt.figure(figsize=(10, 6))  
plt.scatter(y[:, 0], y[:, 1], marker='o') ❶  
plt.xlabel('Первый')  
plt.ylabel('Второй')  
plt.title('Диаграмма рассеяния');
```

- ❶ Построение диаграммы рассеяния с помощью функции `plt.scatter()`.



Рис. 7.14. Диаграмма рассеяния, созданная функцией `plt.scatter()`

Среди прочего функция `plt.scatter()` позволяет добавить третье измерение, описываемое цветовой полосой, которая отображается рядом с областью построения. На рис. 7.15 показана диаграмма рассеяния с цветовой идентификацией точек данных и легендой в виде градиентной полосы в правой части. Для этого генерируется третий набор случайных значений, на этот раз состоящий из целых чисел в диапазоне от 0 до 10.

```
In [27]: c = np.random.randint(0, 10, len(y))
```

```
In [28]: plt.figure(figsize=(10, 6))
plt.scatter(y[:, 0], y[:, 1],
            c=c, ❶
            cmap='coolwarm', ❷
            marker='o') ❸
plt.colorbar()
plt.xlabel('Первый')
plt.ylabel('Второй')
plt.title('Диаграмма рассеяния');
```

- ❶ Добавление еще одного ряда данных.
- ❷ Выбор цветовой карты.
- ❸ Выбор маркера в виде жирной точки.

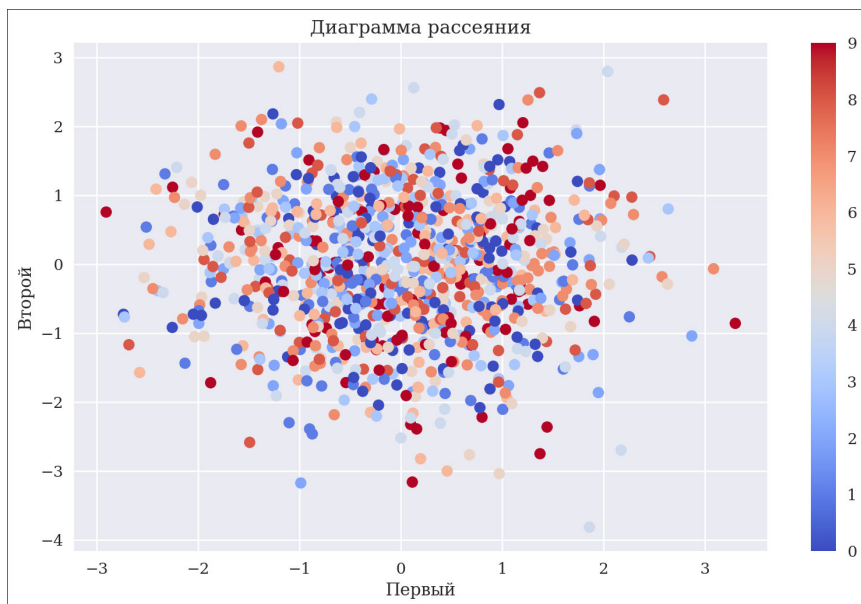


Рис. 7.15. Диаграмма рассеяния с третьим измерением

Следующий популярный тип диаграммы — это *гистограмма*. На рис. 7.16 с помощью гистограммы сопоставляется частотность двух наборов данных.

```
In [29]: plt.figure(figsize=(10, 6))  
plt.hist(y, label=['1-й', '2-й'], bins=25) ❶  
plt.legend(loc=0)  
plt.xlabel('Значение')  
plt.ylabel('Частота')  
plt.title('Гистограмма');
```

❶ Построение гистограммы с помощью функции `plt.hist()`.

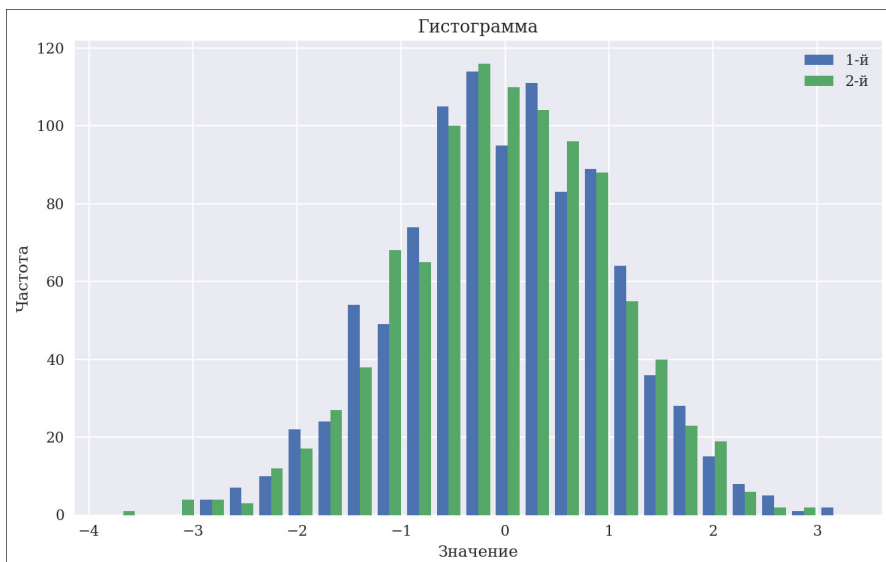


Рис. 7.16. Гистограмма для двух наборов данных

Поскольку гистограммы широко применяются в финансовых приложениях, следует поближе познакомиться с функцией `plt.hist()`. В следующем фрагменте кода перечислены все ее возможные параметры.

```
plt.hist(x, bins=10, range=None, normed=False, weights=None,  
         cumulative=False, bottom=None, histtype='bar', align='mid',  
         orientation='vertical', rwidth=None, log=False, color=None,  
         label=None, stacked=False, hold=None, **kwargs)
```

Основные параметры описаны в табл. 7.5.

Таблица 7.5. Параметры функции `plt.hist()`

Параметр	Описание
<code>x</code>	Массив(ы) в виде списка или объекта <code>ndarray</code>
<code>bins</code>	Количество интервалов разбиения
<code>range</code>	Нижний и верхний пределы построения
<code>normed</code>	Нормирует значения к единичному диапазону
<code>weights</code>	Веса значений <code>x</code>
<code>cumulative</code>	Каждый интервал включает счетчик предыдущих интервалов
<code>histtype</code>	Тип (строковый): <code>bar</code> , <code>barstacked</code> , <code>step</code> , <code>stepfilled</code>
<code>align</code>	Выравнивание (строковый): <code>left</code> , <code>mid</code> , <code>right</code>
<code>orientation</code>	Ориентация (строковый): <code>horizontal</code> , <code>vertical</code>
<code>rwidth</code>	Относительная ширина столбцов
<code>log</code>	Логарифмическая шкала
<code>color</code>	Цвет набора данных (в виде массива)
<code>label</code>	Строка или последовательность строк, представляющих подписи
<code>stacked</code>	Совмещение гистограмм нескольких наборов данных

На рис. 7.17 представлены гистограммы двух предыдущих поднаборов данных, наложенные одна на другую.

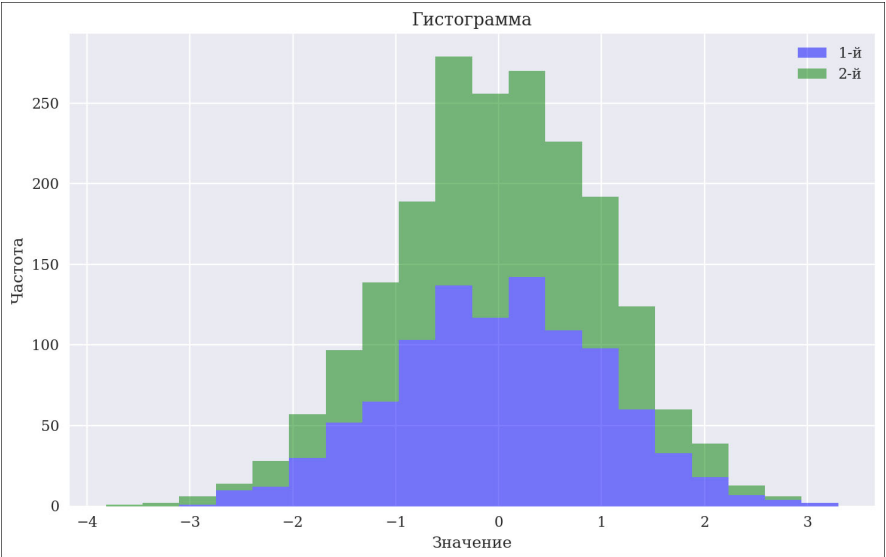


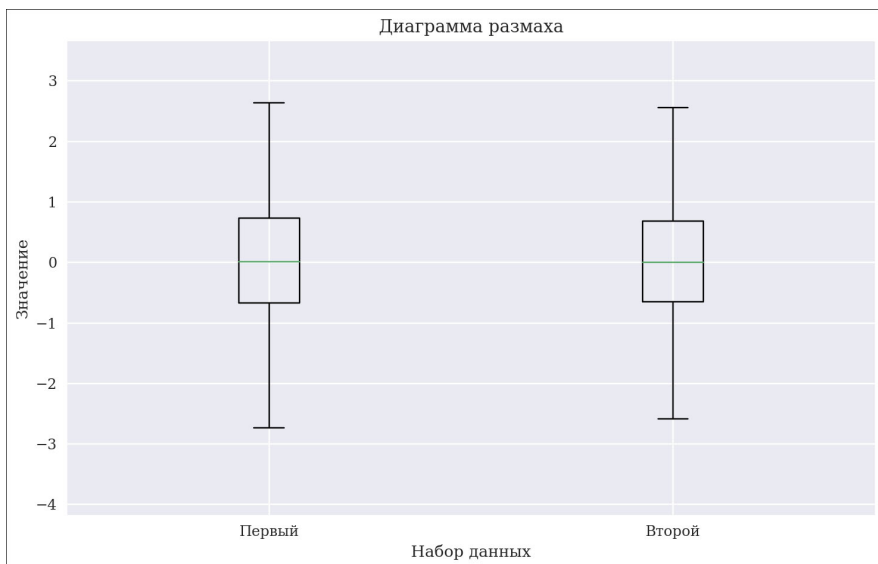
Рис. 7.17. Гистограммы двух наборов данных, наложенные одна на другую

```
In [30]: plt.figure(figsize=(10, 6))
plt.hist(y, label=['1-й', '2-й'], color=['b', 'g'],
         stacked=True, bins=20, alpha=0.5)
plt.legend(loc=0)
plt.xlabel('Значение')
plt.ylabel('Частота')
plt.title('Гистограмма');
```

Еще один полезный вид диаграммы — *диаграмма размаха* (boxplot). Как и гистограмма, она позволяет быстро оценивать основные характеристики наборов данных и сравнивать наборы данных между собой (рис. 7.18).

```
In [31]: fig, ax = plt.subplots(figsize=(10, 6))
plt.boxplot(y) ❶
plt.setp(ax, xticklabels=['Первый', 'Второй']) ❷
plt.xlabel('Набор данных')
plt.ylabel('Значение')
plt.title('Диаграмма размаха');
```

- ❶ Создание диаграммы размаха с помощью функции `plt.boxplot()`.
- ❷ Добавление подписей данных.



*Рис. 7.18. Диаграмма размаха для двух наборов данных*

Здесь также используется функция `plt.setp()`, позволяющая задавать свойства отдельных элементов диаграммы. Пусть, к примеру, график генерируется с помощью следующей строки кода:

```
line = plt.plot(data, 'r')
```

Тогда вот как можно отобразить график в виде штриховой линии:

```
plt.setp(line, linestyle='--')
```

Аналогичным образом можно изменять и другие параметры внешнего вида диаграммы уже после ее создания.

В завершение рассмотрим построение графика математической функции (этот пример также можно найти на сайте <https://matplotlib.org/gallery.html>). В данном примере отображается не только линия графика, но и область под ней, окрашенная другим цветом и ограниченная заданными

значениями. Площадь такой фигуры описывается интегралом вида  $\int_a^b f(x)dx$ , где  $f(x) = \frac{1}{2}e^x + 1$ ,  $a = \frac{1}{2}$ ,  $b = \frac{3}{2}$ . На рис. 7.19 показано, что при построении такого рода графиков пакет `matplotlib` следует правилам форматирования формул, принятым в LaTeX. Сначала определим функцию, зададим пределы интегрирования и сформируем наборы значений  $x$  и  $y$ .

```
In [32]: def func(x):  
         return 0.5 * np.exp(x) + 1 ❶  
         a, b = 0.5, 1.5 ❷  
         x = np.linspace(0, 2) ❸  
         y = func(x) ❹  
         Ix = np.linspace(a, b) ❺  
         Iy = func(Ix) ❻  
         verts = [(a, 0)] + list(zip(Ix, Iy)) + [(b, 0)] ❼
```

- ❶ Определение функции.
- ❷ Пределы интегрирования.
- ❸ Значения  $x$  для графика функции.
- ❹ Значения  $y$  для графика функции.
- ❺ Значения  $x$  в пределах интегрирования.
- ❻ Значения  $y$  в пределах интегрирования.
- ❼ Список кортежей, которые представляют координаты отображаемой области интегрирования.



Теперь напишем код построения графика, который в данном случае получается более сложным, поскольку приходится задействовать много объектов.

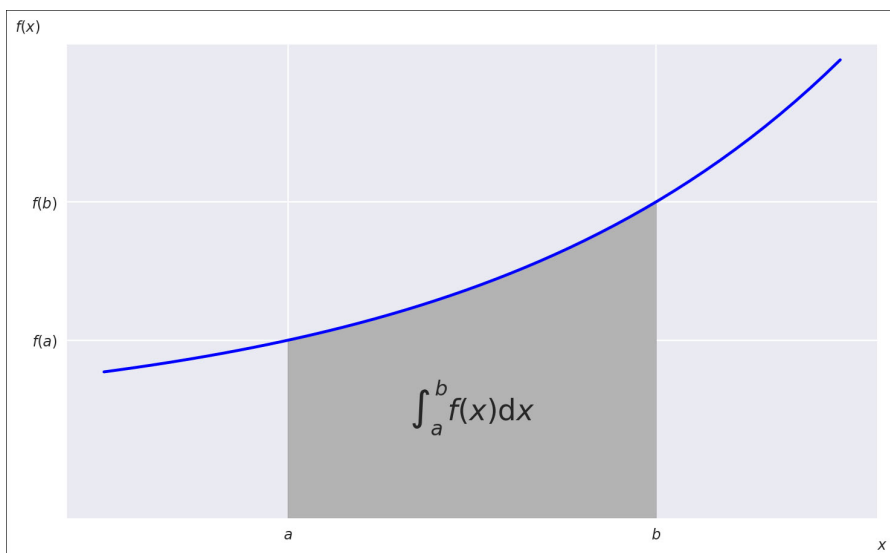


Рис. 7.19. Экспоненциальная функция, область интегрирования и надписи в стиле LaTeX

```
In [33]: from matplotlib.patches import Polygon
fig, ax = plt.subplots(figsize=(10, 6))
plt.plot(x, y, 'b', linewidth=2) ❶
plt.ylim(bottom=0) ❷
poly = Polygon(verts, facecolor='0.7', edgecolor='0.5') ❸
ax.add_patch(poly) ❹
plt.text(0.5 * (a + b), 1, r'$\int_a^b f(x)\mathrm{d}x$',
        horizontalalignment='center', fontsize=20) ❺
plt.figtext(0.9, 0.075, '$x$') ❻
plt.figtext(0.075, 0.9, '$f(x)$') ❼
ax.set_xticks((a, b)) ❽
ax.set_xticklabels(('a', 'b')) ❾
ax.set_yticks([func(a), func(b)]) ❿
ax.set_yticklabels(('f(a)', 'f(b)')); ⓫
```

- ❶ Построение графика функции, представленного синей линией.
- ❷ Определение минимального значения  $y$  на оси ординат.
- ❸ Окрашивание области интегрирования (под графиком) серым цветом.

- ❹ Добавление формулы интегрирования в область построения.
- ❺ Добавление подписей осей.
- ❻ Добавление надписей на оси X.
- ❼ Добавление надписей на оси Y.

## Статические трехмерные диаграммы

В финансовом анализе трехмерные визуализации встречаются не слишком часто. Одна из областей их применения — построение поверхностей волатильности, описывающих прогнозируемую волатильность одновременно для целого ряда сроков исполнения и страйк-цен торгуемых опционов (в приложении Б приведено несколько примеров такого рода визуализаций). В следующем примере строится диаграмма, имитирующая поверхность волатильности. Здесь используются следующие финансовые параметры:

- страйк-цена опциона — от 50 до 150;
- срок исполнения опциона — от 0,5 до 2,5 лет.

Это формирует двухмерную систему координат, которая строится с помощью функции `np.meshgrid()` пакета NumPy, получающей в качестве аргументов два одномерных массива `ndarray`.

```
In [34]: strike = np.linspace(50, 150, 24) ❶
```

```
In [35]: ttm = np.linspace(0.5, 2.5, 24) ❷
```

```
In [36]: strike, ttm = np.meshgrid(strike, ttm) ❸
```

```
In [37]: strike[:2].round(1) ❹
```

```
Out[37]: array([[ 50. ,  54.3,  58.7,  63. ,  67.4,  71.7,  76.1,  80.4,
                  84.8,  89.1,  93.5,  97.8, 102.2, 106.5, 110.9,
                  115.2, 119.6, 123.9, 128.3, 132.6, 137. , 141.3,
                  145.7, 150. ],
                [ 50. ,  54.3,  58.7,  63. ,  67.4,  71.7,  76.1,  80.4,
                  84.8,  89.1,  93.5,  97.8, 102.2, 106.5, 110.9,
                  115.2, 119.6, 123.9, 128.3, 132.6, 137. , 141.3,
                  145.7, 150. ]])
```

```
In [38]: iv = (strike - 100) ** 2 / (100 * strike) / ttm ❺
```

```
In [39]: iv[:5, :3] ❹
Out[39]: array([[1.          , 0.76695652, 0.58132045],
                [0.85185185, 0.65333333, 0.4951989 ],
                [0.74193548, 0.56903226, 0.43130227],
                [0.65714286, 0.504      , 0.38201058],
                [0.58974359, 0.45230769, 0.34283001]])
```

- ❶ Массив страйк-цен.
- ❷ Массив сроков исполнения.
- ❸ Создание двух двумерных объектов ndarray.
- ❹ Вымышленные данные прогнозируемой волатильности.

В следующем фрагменте кода строится график, показанный на рис. 7.20.

```
In [40]: from mpl_toolkits.mplot3d import Axes3D ❶
fig = plt.figure(figsize=(11, 8))
ax = fig.gca(projection='3d') ❷
surf = ax.plot_surface(strike, ttm, iv, rstride=2,
                      cstride=2, cmap=plt.cm.coolwarm,
                      linewidth=0.5, antialiased=True) ❸
ax.set_xlabel('\nСтрайк-цена') ❹
ax.set_ylabel('\nСрок исполнения') ❺
ax.set_zlabel('\nОжидаемая волатильность') ❻
fig.colorbar(surf, shrink=0.5, aspect=5); ❼
```

- ❶ Импорт настроек трехмерного графика из модуля Axes3D, который применяется неявно.
- ❷ Настройка области построения трехмерного графика.
- ❸ Создание трехмерного графика.
- ❹ Добавление подписи оси X.
- ❺ Добавление подписи оси Y.
- ❻ Добавление подписи оси Z.
- ❼ Создание цветовой полосы.

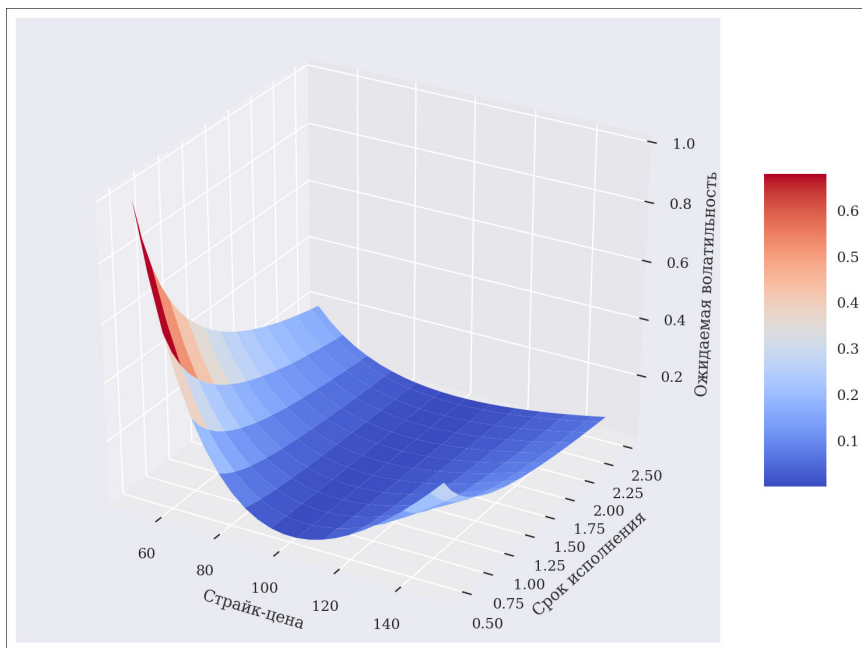


Рис. 7.20. Трехмерная поверхность ожидаемой волатильности для вымышленных данных

В табл. 7.6 описаны различные параметры функции `plt.plot_surface()`.

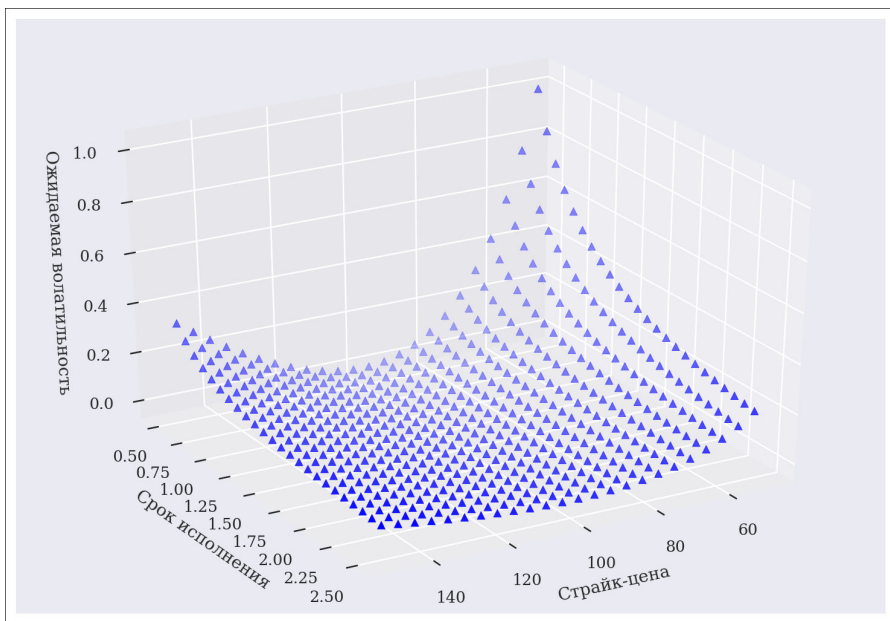
Таблица 7.6. Параметры функции `plt.plot_surface()`

Параметр	Описание
X, Y, Z	Отображаемые данные в виде двухмерных массивов
rstride	Шаг сетки для строк массива
cstride	Шаг сетки для столбцов массива
color	Цвет участков поверхности
cmar	Цветовая карта участков поверхности
facecolors	Цвета отдельных граней участков поверхности
norm	Экземпляр объекта <code>Normalize</code> , применяемый для сопоставления значений с цветами
vmin	Минимальное отображаемое значение
vmax	Максимальное отображаемое значение
shade	Затенение цветов граней

Как и в двухмерных графиках, в трехмерной визуализации данные можно представлять точками или даже, как в следующем примере, треугольными маркерами. На рис. 7.21 строится трехмерная диаграмма рассеяния, только на этот раз под другим углом просмотра, который настраивается с помощью метода `view_init()`.

```
In [41]: fig = plt.figure(figsize=(10, 7))
         ax = fig.add_subplot(111, projection='3d')
         ax.view_init(30, 60) ❶
         ax.scatter(strike, ttm, iv, zdir='z', s=25,
                   c='b', marker='^') ❷
         ax.set_xlabel('\nСтрайк-цена')
         ax.set_ylabel('\nСрок исполнения')
         ax.set_zlabel('\nОжидаемая волатильность');
```

- ❶ Задание угла просмотра.
- ❷ Создание трехмерной диаграммы рассеяния.



*Рис. 7.21. Трехмерная диаграмма рассеяния, описывающая прогнозируемую волатильность для вымышленных данных*

# Интерактивные двухмерные диаграммы

Пакет `matplotlib` позволяет создавать диаграммы, сохраняемые в виде статических растровых изображений или файлов формата PDF. Но есть много других библиотек, обеспечивающих построение интерактивных диаграмм, основанных на стандарте D3.js. Такие диаграммы можно масштабировать, инспектировать и т.п. К тому же их легко встраивать на веб-страницы.

Самая популярная платформа такого рода — библиотека `plotly` (<http://plot.ly/>), которая широко применяется для визуализации научных данных. Ее преимуществом является тесная интеграция с экосистемой Python и простота применения, особенно в связке с объектами `DataFrame` пакета `pandas` и оболочечным пакетом `Cufflinks` (<http://github.com/santosjorge/cufflinks>).

Для доступа к определенным функциям библиотеки `plotly` необходимо пройти процедуру бесплатной регистрации (<https://chart-studio.plotly.com/Auth/login/#/>). После получения учетных данных необходимо сохранить их локально, чтобы иметь возможность использовать на постоянной основе (подробнее об этом — на сайте <https://plot.ly/python/getting-started/>).

В этом разделе мы применим пакет `Cufflinks` для построения интерактивных диаграмм на основе данных, сохраненных в объектах `DataFrame`.

## Базовые графики

Чтобы начать работать с пакетом `Cufflinks` в среде Jupyter Notebook, необходимо импортировать требуемые модули, а затем перейти в *режим блокнота*.

```
In [42]: import pandas as pd
```

```
In [43]: import cufflinks as cf ❶
```

```
In [44]: import plotly.offline as plyo ❷
```

```
In [45]: plyo.init_notebook_mode(connected=True) ❸
```

- ❶ Импорт пакета `Cufflinks`.
- ❷ Импорт автономных инструментов построения графиков библиотеки `plotly`.
- ❸ Включение режима блокнота.



## Дистанционная или локальная визуализация

Существует возможность визуализации диаграмм на серверах `plotly`. Но в режиме ноутбука диаграммы строятся значительно быстрее, особенно при обработке больших массивов данных. Тем не менее некоторые функции библиотеки `plotly`, например служба потоковой визуализации, доступны только при подключении к соответствующему серверу.

В следующем примере снова используется набор данных, представленный псевдослучайными числами, только на этот раз они хранятся в объекте `DataFrame` и снабжены индексами `DatetimeIndex` (временной ряд).

```
In [46]: a = np.random.standard_normal((250, 5)).cumsum(axis=0) ❶
```

```
In [47]: index = pd.date_range('2019-1-1', ❷  
                                freq='B', ❸  
                                periods=len(a)) ❹
```

```
In [48]: df = pd.DataFrame(100 + 5 * a, ❺  
                           columns=list('abcde'), ❻  
                           index=index) ❼
```

```
In [49]: df.head() ❽
```

Out[49]:

	a	b	c	d	e
2019-01-01	109.037535	98.693865	104.474094	96.878857	100.621936
2019-01-02	107.598242	97.005738	106.789189	97.966552	100.175313
2019-01-03	101.639668	100.332253	103.183500	99.747869	107.902901
2019-01-04	98.500363	101.208283	100.966242	94.023898	104.387256
2019-01-07	93.941632	103.319168	105.674012	95.891062	86.547934

- ❶ Последовательность псевдослучайных чисел со стандартным нормальным распределением.
- ❷ Начальная дата для объекта `DatetimeIndex`.
- ❸ Частота (рабочие дни).
- ❹ Количество периодов разбиения.
- ❺ Линейное преобразование исходных данных.
- ❻ Односимвольные заголовки столбцов.
- ❼ Объект `DatetimeIndex`.
- ❽ Первые пять строк данных.

Пакет Cufflinks добавляет в класс `DataFrame` метод `df.iplot()`, который использует библиотеку `plotly` в качестве сервера построения интерактивных диаграмм. В примерах данного раздела поддерживается возможность загрузки диаграммы в виде статического растрового файла, который в свою очередь можно встроить на веб-страницу. Но в среде `Jupyter Notebook` все создаваемые диаграммы являются интерактивными. Результат выполнения следующего фрагмента кода показан на рис. 7.22.

```
In [50]: plyo.iplot( ❶  
                df.iplot(asFigure=True), ❷  
                image='png', ❸  
                filename='ply_01' ❹  
            )
```

- ❶ Подключение автономных (режим блокнота) инструментов библиотеки `plotly`.
- ❷ Передача аргумента `asFigure=True` методу `df.iplot()` для перехода в режим построения встраиваемых диаграмм.
- ❸ Параметр `image` позволяет получить статическую версию диаграммы.
- ❹ Указание имени файла для растровой диаграммы (расширение добавляется автоматически).



Рис. 7.22. Графики временных рядов, созданные средствами `plotly`, `pandas` и `Cufflinks`



Для создаваемых диаграмм поддерживается множество параметров настройки (рис. 7.23).



Рис. 7.23. Графики двух рядов данных, хранящихся в объекте *DataFrame*

```
In [51]: plyo.ipplot(
    df[['a', 'b']].ipplot(asFigure=True,
        theme='polar', ❶
        title='График временных рядов', ❷
        xTitle='Дата', ❸
        yTitle='Значение', ❹
        mode={'a': 'markers', 'b': 'lines+markers'}, ❺
        symbol={'a': 'circle', 'b': 'diamond'}, ❻
        size=3.5, ❼
        colors={'a': 'blue', 'b': 'magenta'}, ❽
    ),
    image='png',
    filename='ply_02'
)
```

- ❶ Выбор шаблона (стиля) диаграммы.
- ❷ Добавление заголовка диаграммы.

- ③ Добавление подписи оси X.
- ④ Добавление подписи оси Y.
- ⑤ Определение *режима* отображения значений столбцов (линии, маркеры и т.п.).
- ⑥ Определение символов, используемых в качестве маркеров.
- ⑦ Фиксирование размера маркеров.
- ⑧ Определение цветов для значений столбцов.

Подобно пакету `matplotlib`, библиотека `plotly` позволяет строить диаграммы самого разного типа. В пакете `Cufflinks` поддерживаются следующие типы диаграмм: `chart`, `scatter`, `bar`, `box`, `spread`, `ratio`, `heatmap`, `surface`, `histogram`, `bubble`, `bubble3d`, `scatter3d`, `scattergeo`, `ohlc`, `candle`, `pie` и `choropleth`. В качестве примера рассмотрим, как построить гистограмму (рис. 7.24).



Рис. 7.24. Гистограммы для каждого столбца значений, хранящихся в объекте `DataFrame`

```
In [52]: plyo.iplot(
          df.iplot(kind='hist', ①
                  subplots=True, ②
```

```

        bins=15, ❸
        asFigure=True),
    image='png',
    filename='ply_03'
)

```

- ❶ Определение типа диаграммы.
- ❷ Отображение каждого столбца значений в отдельной области построения.
- ❸ Установка параметра `bins` (количество столбцов гистограммы).

## Финансовые диаграммы

Совместное использование инструментов `plotly`, `pandas` и `Cufflinks` дает особенно много преимуществ при работе с финансовыми временными рядами. Например, пакет `Cufflinks` содержит готовые функции для создания популярных финансовых диаграмм и отображения на них специальных финансовых индикаторов, таких как RSI (Relative Strength Index — индекс относительной силы). С этой целью создается постоянный объект `QuantFig`, данные которого визуализируются на диаграмме так же, как и данные объекта `DataFrame`.

В следующем примере используется финансовый набор данных — временной ряд валютного курса EUR/USD (источник: FXCM Forex Capital Markets Ltd.).

```
In [54]: raw = pd.read_csv('../source/fxcm_eur_usd_eod_data.csv',
                           index_col=0, parse_dates=True) ❶
```

```
In [55]: raw.info() ❷
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1547 entries, 2013-01-01 22:00:00 to
                2017-12-31 22:00:00
Data columns (total 8 columns):
BidOpen      1547 non-null float64
BidHigh      1547 non-null float64
BidLow       1547 non-null float64
BidClose     1547 non-null float64
AskOpen      1547 non-null float64
AskHigh      1547 non-null float64
AskLow       1547 non-null float64
AskClose     1547 non-null float64
```

```
dtypes: float64(8)
memory usage: 108.8 KB
```

```
In [56]: quotes = raw[['AskOpen', 'AskHigh', 'AskLow', 'AskClose']] ❸
quotes = quotes.iloc[-60:] ❹
quotes.tail() ❺
```

```
Out[56]:
```

	AskOpen	AskHigh	AskLow	AskClose
2017-12-25 22:00:00	1.18667	1.18791	1.18467	1.18587
2017-12-26 22:00:00	1.18587	1.19104	1.18552	1.18885
2017-12-27 22:00:00	1.18885	1.19592	1.18885	1.19426
2017-12-28 22:00:00	1.19426	1.20256	1.19369	1.20092
2017-12-31 22:00:00	1.20092	1.20144	1.19994	1.20144

- ❶ Считывание финансовых данных из CSV-файла.
- ❷ Полученный объект `DataFrame` содержит много столбцов и более 1500 строк данных.
- ❸ Выбор 4 столбцов из объекта `DataFrame` (формат “Open-High-Low-Close”, или OHLC).
- ❹ Выборка подмножества данных объекта `DataFrame`.
- ❺ Отображение последних пяти строк полученного набора данных.

В процессе инициализации объект `QuantFig` получает данные объекта `DataFrame` и выполняет ряд базовых настроек. Визуализация данных, хранящихся в объекте `QuantFig`, осуществляется с помощью метода `qf.iplot()` (рис. 7.25).

```
In [57]: qf = cf.QuantFig(
            quotes, ❶
            title='Валютный курс пары EUR/USD', ❷
            legend='top', ❸
            name='EUR/USD' ❹
        )
```

```
In [58]: ptyo.iplot(
            qf.iplot(asFigure=True),
            image='png',
            filename='qf_01'
        )
```

- ❶ Передача объекта `DataFrame` конструктору класса `QuantFig`.
- ❷ Добавление заголовка диаграммы.
- ❸ Размещение легенды в верхней части диаграммы.
- ❹ Название набора данных.



*Рис. 7.25. Диаграмма OHLC для валютной пары EUR/USD*

Объект `QuantFig` поддерживает различные методы, позволяющие добавить на диаграмму специальные финансовые обозначения, например линии Боллинджера (рис. 7.26).

```
In [59]: qf.add_bollinger_bands(periods=15, ❶
                                boll_std=2) ❷
```

```
In [60]: pty.ipplot(qf.ipplot(asFigure=True),
                    image='png',
                    filename='qf_02'
                    )
```

- ❶ Количество периодов для линий Боллинджера.
- ❷ Количество среднеквадратических отклонений, применяемых при вычислении линий Боллинджера.



*Рис. 7.26. Добавление линий Боллинджера на диаграмму OHLC для валютной пары EUR/USD*

Кроме того, на такие диаграммы можно добавлять вспомогательные индикаторы, например RSI (Relative Strength Index — индекс относительной силы), выводимые в отдельной области построения (рис. 7.27).

```
In [61]: qf.add_rsi( periods=14, ❶
              showbands=False) ❷

In [62]: plyo.iplot(
            qf.iplot(asFigure=True),
            image='png',
            filename='qf_03'
        )
```

- 1 Задание числа периодов для индикатора RSI.
- 2 Подавление вывода нижнего и верхнего пределов.



Рис. 7.27. Добавление линий Боллинджера и индикатора RSI на диаграмму OHLC для валютной пары EUR/USD

## Резюме

Пакет `matplotlib` считается стандартным средством визуализации данных в Python. Он тесно интегрирован с пакетами `NumPy` и `pandas`, а его базовые функции просты и удобны. В то же время программный интерфейс пакета достаточно обширен и сложен, из-за чего нереально охватить все его возможности в рамках одной главы.

В этой главе были описаны основные функции пакета `matplotlib`, применяемые для создания двух- и трехмерных финансовых диаграмм. Дополнительные примеры визуализации будут приведены в других главах.

Кроме того, в главе рассматривалась библиотека `plotly`, которая в связке с пакетом `Cufflinks` позволяет легко создавать интерактивные диаграммы формата D3.js, поскольку для этого достаточно вызвать один-единственный метод

объекта `DataFrame`, а всеми деталями визуализации управляет сервер. Кроме того, в пакете `Cufflinks` имеется также объект `QuantFig`, который упрощает построение стандартных финансовых диаграмм, отображающих наиболее важные финансовые индикаторы.

## Дополнительные ресурсы

Дополнительная информация о пакете `matplotlib` доступна на следующих сайтах:

- домашняя страница официального сайта пакета, которая послужит хорошей отправной точкой (<https://matplotlib.org/index.html>);
- галерея примеров (<https://matplotlib.org/gallery.html>);
- руководство по созданию двухмерных диаграмм ([https://matplotlib.org/users/pyplot\\_tutorial.html](https://matplotlib.org/users/pyplot_tutorial.html));
- руководство по созданию трехмерных диаграмм ([https://matplotlib.org/mpl\\_toolkits/mplot3d/tutorial.html](https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html)).

Галерея примеров — самый удобный ресурс для выбора нужного способа визуализации данных, поскольку для каждого типа диаграммы сразу же предлагается готовый код.

Информация о библиотеке `plotly` и пакете `Cufflinks` доступна на следующих сайтах:

- официальный сайт проекта `plotly` (<https://plot.ly>);
- руководство по использованию библиотеки `plotly` в проектах Python (<https://plot.ly/python/getting-started>);
- страница пакета `Cufflinks` на сайте GitHub (<https://github.com/santosjorge/cufflinks>).





## Финансовые временные ряды

Время мешает всему происходить одновременно.

*Рэй Каммингс*

Временные ряды играют важную роль в финансовом анализе. Это наборы данных, которые индексируются по дате и/или времени. В качестве примера можно привести данные об изменении стоимости акций. Колебания курса доллара к евро тоже представляют собой временной ряд. Текущий валютный курс соответствует короткому промежутку времени, а совокупность таких котировок образует временной ряд.

Нет такой финансовой дисциплины, в которой не учитывался бы фактор времени. В финансовом анализе оно играет не менее важную роль, чем в физике и других точных науках. В Python основной инструмент для работы с временными рядами — библиотека `pandas`. Уэс Маккинни, автор и основной разработчик библиотеки, приступил к ее созданию, работая аналитиком в AQR Capital Management, одном из крупнейших хедж-фондов. Таким образом, можно смело утверждать, что библиотека `pandas` с самого начала предназначалась для работы с финансовыми временными рядами.

В главе рассматриваются следующие темы.

### *Финансовые данные*

В этом разделе вы познакомитесь с общими принципами обработки финансовых временных рядов с помощью библиотеки `pandas` и научитесь выполнять основные операции, такие как импорт данных, получение статистической сводки, вычисление изменений во времени и прореживание данных.

### *Скольльзящая статистика*

В финансовом анализе скользящая статистика играет важную роль. Это статистический показатель, вычисляемый за фиксированный промежуток времени с перемещением вперед по всему временному ряду. Наиболее популярный пример — скользящее среднее. Из этого раздела вы узнаете о том, как вычислить такую статистику с помощью библиотеки `pandas`.

### Корреляционный анализ

В этом разделе мы рассмотрим пример работы с временными рядами на основе фондового индекса S&P 500 и индекса волатильности VIX. Мы постараемся подтвердить тот (эмпирический) факт, что между двумя индексами существует отрицательная корреляция.

### Высокочастотные данные

Этот раздел посвящен высокочастотным, или *тиковым*, данным, которые широко применяются в финансовом анализе. В библиотеке `pandas` имеются удобные средства для работы с такими данными.

## Финансовые данные

В этом разделе мы будем работать с локально хранящимися финансовыми данными, представленными в формате CSV. С технической точки зрения CSV-файл представляет собой простой текстовый файл с построчной структурой, в которой отдельные значения разделяются запятыми. Прежде чем импортировать такой файл, необходимо подключить несколько пакетов и задать ряд настроек.

```
In [1]: import numpy as np
import pandas as pd
from pylab import mpl, plt
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

### Импорт данных

Для импорта финансовых данных, сохраненных в наиболее распространенных форматах (CSV, SQL, Excel и др.), в библиотеке `pandas` предусмотрен целый ряд функций и методов объекта `DataFrame`. В следующем примере с помощью функции `pd.read_csv()` импортируется временной ряд, хранящийся в CSV-файле<sup>1</sup>.

```
In [2]: filename = '../..source/tr_eikon_eod_data.csv' ❶
```

```
In [3]: f = open(filename, 'r') ❷
f.readlines()[:5] ❷
```

---

<sup>1</sup> В файле хранятся данные о стоимости различных финансовых инструментов на конец торгов (EOD), предоставляемые службой Thomson Reuters Eikon Data.

```
Out[3]: ['Date,AAPL.O,MSFT.O,INTC.O,AMZN.O,GS.N,SPY,.SPX,.VIX,EUR=,
XAU=,GDZ,,GLD\n',
'2010-01-01,,,,,,1.4323,1096.35,,\n',
'2010-01-04,30.57282657,30.95,20.88,133.9,173.08,113.33,
1132.99,20.04,,1.4411,1120.0,47.71,109.8\n',
'2010-01-05,30.625683660000004,30.96,20.87,134.69,176.14,
113.63,1136.52,,19.35,1.4368,1118.65,48.17,109.7\n',
'2010-01-06,30.1385412900000003,30.77,20.8,132.25,174.26,
113.71,1137.14,,19.16,1.4412,1138.5,49.34,111.51\n']
```

```
In [4]: data = pd.read_csv(filename, ❸
                                index_col=0, ❹
                                parse_dates=True) ❺
```

```
In [5]: data.info() ❻
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
Data columns (total 12 columns):
AAPL.O      2138 non-null float64
MSFT.O      2138 non-null float64
INTC.O      2138 non-null float64
AMZN.O      2138 non-null float64
GS.N        2138 non-null float64
SPY         2138 non-null float64
.SPX        2138 non-null float64
.VIX        2138 non-null float64
EUR=        2216 non-null float64
XAU=        2211 non-null float64
GDZ         2138 non-null float64
GLD         2138 non-null float64
dtypes: float64(12)
memory usage: 225.1 KB
```

- ❶ Задание пути к файлу.
- ❷ Считывание первых пять строк необработанных данных (Linux/Mac).
- ❸ Передача имени файла в функцию `pd.read_csv()`.
- ❹ Выбор индексов из первого столбца.
- ❺ Назначение индексам типа данных `datetime`.
- ❻ Полученный объект `DataFrame`.

На этом этапе можно просмотреть имеющийся набор данных или построить базовые графики (рис. 8.1).

In [6]: data.head() ❶

Out[6]:

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	30.572827	30.950	20.88	133.90	173.08	113.33
2010-01-05	30.625684	30.960	20.87	134.69	176.14	113.63
2010-01-06	30.138541	30.770	20.80	132.25	174.26	113.71
2010-01-07	30.082827	30.452	20.60	130.00	177.67	114.19

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2010-01-01	NaN	NaN	1.4323	1096.35	NaN	NaN
2010-01-04	1132.99	20.04	1.4411	1120.00	47.71	109.80
2010-01-05	1136.52	19.35	1.4368	1118.65	48.17	109.70
2010-01-06	1137.14	19.16	1.4412	1138.50	49.34	111.51
2010-01-07	1141.69	19.06	1.4318	1131.90	49.10	110.82

In [7]: data.tail() ❷

Out[7]:

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
Date						
2018-06-25	182.17	98.39	50.71	1663.15	221.54	271.00
2018-06-26	184.43	99.08	49.67	1691.09	221.58	271.60
2018-06-27	184.16	97.54	48.76	1660.51	220.18	269.35
2018-06-28	185.50	98.63	49.25	1701.45	223.42	270.89
2018-06-29	185.11	98.61	49.71	1699.80	220.57	271.28

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2018-06-25	2717.07	17.33	1.1702	1265.00	22.01	119.89
2018-06-26	2723.06	15.92	1.1645	1258.64	21.95	119.26
2018-06-27	2699.63	17.91	1.1552	1251.62	21.81	118.58
2018-06-28	2716.31	16.85	1.1567	1247.88	21.93	118.22
2018-06-29	2718.37	16.09	1.1683	1252.25	22.31	118.65

In [8]: data.plot(figsize=(10, 12), subplots=True); ❸

- ❶ Отображение первых пяти...
- ❷ ...и последних пяти строк.
- ❸ Визуализация полученных наборов данных в разных областях построения.

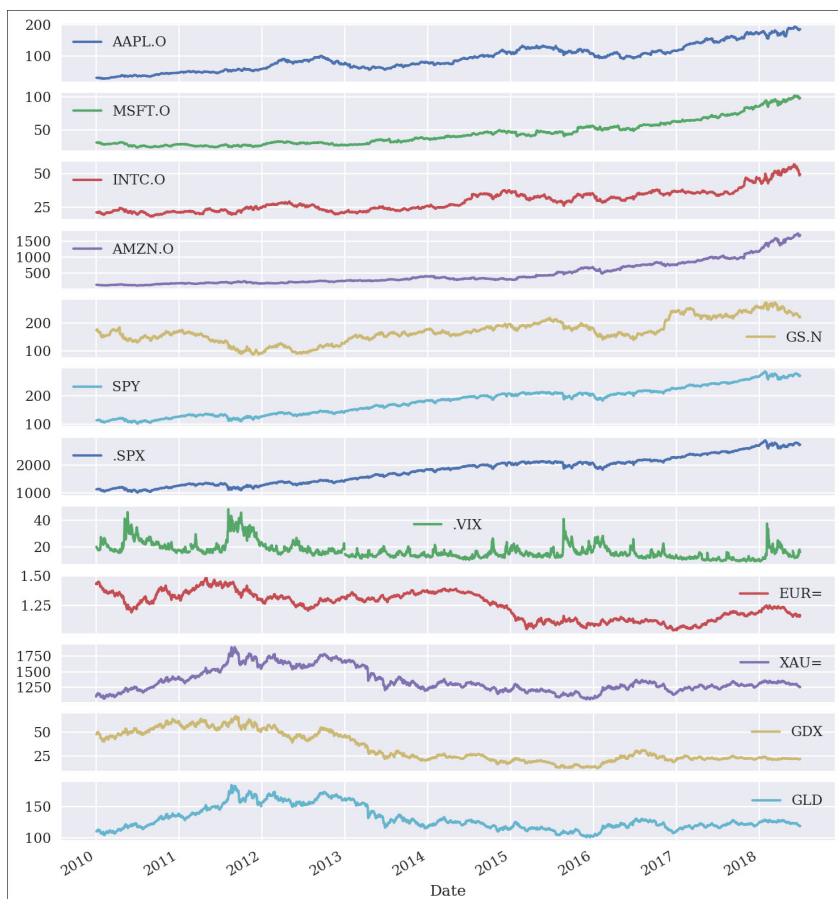


Рис. 8.1. Графики финансовых временных рядов

Используемые нами данные получены с помощью программного интерфейса Thomson Reuters Eikon Data. В терминологии разработчиков коды, которыми обозначаются финансовые инструменты, называются *символьными идентификаторами Reuters* (Reuters Instrument Code — RIC). Мы будем работать со следующими финансовыми инструментами.

```
In [9]: instruments = ['Apple Stock', 'Microsoft Stock',
                      'Intel Stock', 'Amazon Stock',
                      'Goldman Sachs Stock',
                      'SPDR S&P 500 ETF Trust',
                      'S&P 500 Index', 'VIX Volatility Index',
```

```
'EUR/USD Exchange Rate', 'Gold Price',
'VanEck Vectors Gold Miners ETF',
'SPDR Gold Trust']
```

```
In [10]: for ric, name in zip(data.columns, instruments):
        print('{:8s} | {}'.format(ric, name))
AAPL.O   | Apple Stock
MSFT.O   | Microsoft Stock
INTC.O   | Intel Stock
AMZN.O   | Amazon Stock
GS.N     | Goldman Sachs Stock
SPY      | SPDR S&P 500 ETF Trust
.SPX     | S&P 500 Index
.VIX     | VIX Volatility Index
EUR=     | EUR/USD Exchange Rate
XAU=     | Gold Price
GDx      | VanEck Vectors Gold Miners ETF
GLD      | SPDR Gold Trust
```

## Статистическая сводка

Следующим этапом анализа финансовых данных будет получение сводной статистики, что позволяет выявить скрытые закономерности в наборе.

```
In [11]: data.info() ❶
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
Data columns (total 12 columns):
AAPL.O    2138 non-null float64
MSFT.O    2138 non-null float64
INTC.O    2138 non-null float64
AMZN.O    2138 non-null float64
GS.N      2138 non-null float64
SPY       2138 non-null float64
.SPX      2138 non-null float64
.VIX      2138 non-null float64
EUR=      2216 non-null float64
XAU=      2211 non-null float64
GDx       2138 non-null float64
GLD       2138 non-null float64
dtypes: float64(12)
memory usage: 225.1 KB
```

In [12]: data.describe().round(2) ②

Out[12]:

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
count	2138.00	2138.00	2138.00	2138.00	2138.00	2138.00
mean	93.46	44.56	29.36	480.46	170.22	180.32
std	40.55	19.53	8.17	372.31	42.48	48.19
min	27.44	23.01	17.66	108.61	87.70	102.20
25%	60.29	28.57	22.51	213.60	146.61	133.99
50%	90.55	39.66	27.33	322.06	164.43	186.32
75%	117.24	54.37	34.71	698.85	192.13	210.99
max	193.98	102.49	57.08	1750.08	273.38	286.58

	.SPX	.VIX	EUR=	XAU=	GDJ	GLD
count	2138.00	2138.00	2216.00	2211.00	2138.00	2138.00
mean	1802.71	17.03	1.25	1349.01	33.57	130.09
std	483.34	5.88	0.11	188.75	15.17	18.78
min	1022.58	9.14	1.04	1051.36	12.47	100.50
25%	1338.57	13.07	1.13	1221.53	22.14	117.40
50%	1863.08	15.58	1.27	1292.61	25.62	124.00
75%	2108.94	19.07	1.35	1428.24	48.34	139.00
max	2872.87	48.00	1.48	1898.99	66.63	184.59

- ① Функция `info()` возвращает метаданные об объекте `DataFrame`.
- ② Функция `describe()` возвращает статистические показатели по столбцам.



### Быстрый анализ

В библиотеке `pandas` имеется несколько полезных методов, позволяющих получить быструю информацию о финансовых временных рядах, в частности `info()` и `describe()`. С их помощью можно также убедиться в том, что процедура импорта прошла корректно (например, действительно ли объект `DataFrame` содержит индекс типа `DatetimeIndex`).

Есть также возможность выбрать конкретную статистику.

In [13]: data.mean() ①

Out[13]: AAPL.O 93.455973  
MSFT.O 44.561115  
INTC.O 29.364192  
AMZN.O 480.461251  
GS.N 170.216221  
SPY 180.323029



```
.SPX    1802.713106
.VIX     17.027133
EUR=     1.248587
XAU=    1349.014130
GDG      33.566525
GLD     130.086590
dtype: float64
```

```
In [14]: data.aggregate([min, ②
                        np.mean, ③
                        np.std, ④
                        np.median, ⑤
                        max] ⑥
                    ).round(2)
```

Out[14]:

	APL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
<b>min</b>	27.44	23.01	17.66	108.61	87.70	102.20
<b>mean</b>	93.46	44.56	29.36	480.46	170.22	180.32
<b>std</b>	40.55	19.53	8.17	372.31	42.48	48.19
<b>median</b>	90.55	39.66	27.33	322.06	164.43	186.32
<b>max</b>	193.98	102.49	57.08	1750.08	273.38	286.58

	.SPX	.VIX	EUR=	XAU=	GDG	GLD
<b>min</b>	1022.58	9.14	1.04	1051.36	12.47	100.50
<b>mean</b>	1802.71	17.03	1.25	1349.01	33.57	130.09
<b>std</b>	483.34	5.88	0.11	188.75	15.17	18.78
<b>median</b>	1863.08	15.58	1.27	1292.61	25.62	124.00
<b>max</b>	2872.87	48.00	1.48	1898.99	66.63	184.59

- ① Среднее по столбцу.
- ② Минимальное значение столбца.
- ③ Среднее значение столбца.
- ④ Среднеквадратическое отклонение по столбцу.
- ⑤ Медиана столбца.
- ⑥ Максимальное значение столбца.

## Изменения во времени

Методы статистического анализа часто применяются для отслеживания изменений во времени, а не абсолютных значений. Существует несколько способов вычислить изменения, накапливаемые во временном ряде, включая такие статистические показатели, как абсолютная разность, процентное изменение и логарифмическая доходность.

Для вычисления абсолютных разностей в библиотеке `pandas` предусмотрен специальный метод.

```
In [15]: data.diff().head() ❶
```

```
Out[15]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-05	0.052857	0.010	-0.01	0.79	3.06	0.30
2010-01-06	-0.487142	-0.190	-0.07	-2.44	-1.88	0.08
2010-01-07	-0.055714	-0.318	-0.20	-2.25	3.41	0.48

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	NaN	NaN	0.0088	23.65	NaN	NaN
2010-01-05	3.53	-0.69	-0.0043	-1.35	0.46	-0.10
2010-01-06	0.62	-0.19	0.0044	19.85	1.17	1.81
2010-01-07	4.55	-0.10	-0.0094	-6.60	-0.24	-0.69

```
In [16]: data.diff().mean() ❷
```

```
Out[16]: AAPL.O    0.064737
          MSFT.O    0.031246
          INTC.O    0.013540
          AMZN.O    0.706608
          GS.N      0.028224
          SPY       0.072103
          .SPX      0.732659
          .VIX      -0.019583
          EUR=      -0.000119
          XAU=       0.041887
          GDX       -0.015071
          GLD       -0.003455
          dtype: float64
```

- ❶ Метод `diff()` вычисляет абсолютную разность двух индексированных значений.
- ❷ К полученным данным тоже можно применять операции агрегирования.

С точки зрения статистики абсолютная разность — не слишком информативный показатель, поскольку он зависит от масштаба временной шкалы. Как правило, более предпочтительны процентные изменения. В следующем примере вычисляются не только процентные изменения (простая доходность) по всему набору, но и их средние значения для каждого из столбцов (рис. 8.2).

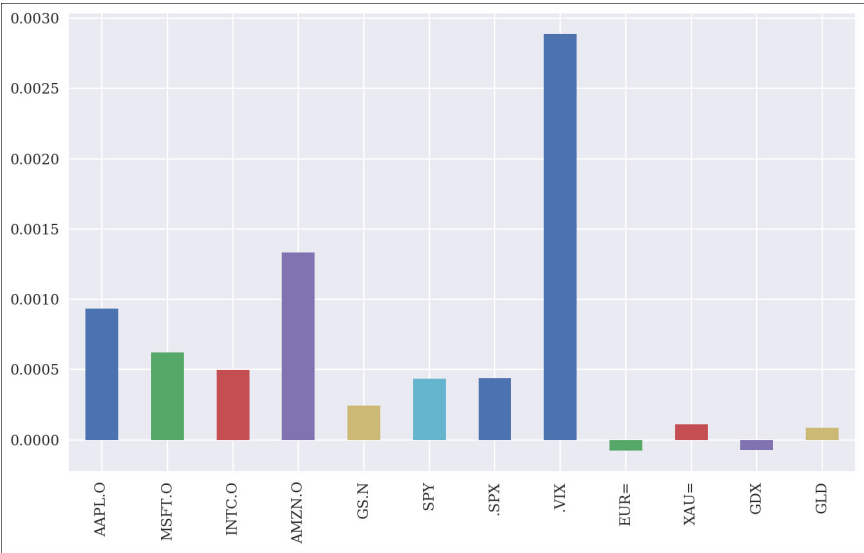


Рис. 8.2. Средние значения процентных изменений

```
In [17]: data.pct_change().round(3).head() ❶
Out[17]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-05	0.002	0.000	-0.000	0.006	0.018	0.003
2010-01-06	-0.016	-0.006	-0.003	-0.018	-0.011	0.001
2010-01-07	-0.002	-0.010	-0.010	-0.017	0.020	0.004

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	NaN	NaN	0.006	0.022	NaN	NaN
2010-01-05	0.003	-0.034	-0.003	-0.001	0.010	-0.001
2010-01-06	0.001	-0.010	0.003	0.018	0.024	0.016
2010-01-07	0.004	-0.005	-0.007	-0.006	-0.005	-0.006

```
In [18]: data.pct_change().mean().plot(kind='bar',
figsize=(10, 6)); ❷
```

- ❶ Метод `pct_change()` вычисляет процентное изменение между двумя индексированными значениями.
- ❷ Визуализация средних значений в виде столбчатой диаграммы.

Помимо простой (процентной) доходности часто вычисляют также логарифмическую доходность. В некоторых финансовых приложениях она оказывается более информативной<sup>2</sup>. На рис. 8.3 показаны графики логарифмической доходности по каждому из рядов. Такой способ визуализации предполагает предварительную *нормализацию* данных.

```
In [19]: rets = np.log(data / data.shift(1)) ❶
```

```
In [20]: rets.head().round(3) ❷
```

```
Out[20]:
```

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-05	0.002	0.000	-0.000	0.006	0.018	0.003
2010-01-06	-0.016	-0.006	-0.003	-0.018	-0.011	0.001
2010-01-07	-0.002	-0.010	-0.010	-0.017	0.019	0.004

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	NaN
2010-01-04	NaN	NaN	0.006	0.021	NaN	NaN
2010-01-05	0.003	-0.035	-0.003	-0.001	0.010	-0.001
2010-01-06	0.001	-0.010	0.003	0.018	0.024	0.016
2010-01-07	0.004	-0.005	-0.007	-0.006	-0.005	-0.006

<sup>2</sup> Одним из преимуществ такого способа оценки является аддитивность во времени, не свойственная простой (процентной) доходности.

In [21]: `rets.cumsum().apply(np.exp).plot(figsize=(10, 6));` ❸

- ❶ Вычисление логарифмической доходности векторизованным способом.
- ❷ Получение выборки.
- ❸ Построение графиков накапливаемой логарифмической доходности: сначала вызывается метод `cumsum()`, а затем к полученным результатам применяется метод `np.exp()`.

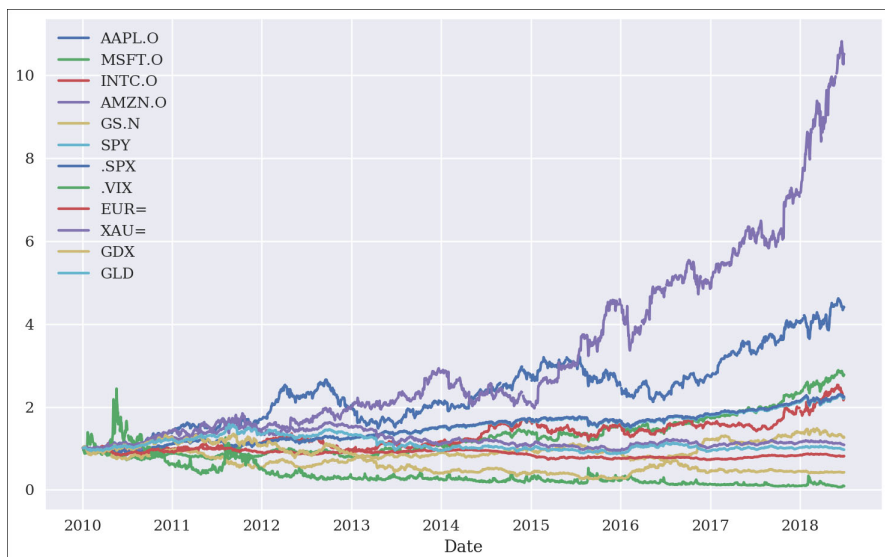


Рис. 8.3. Накапливаемая логарифмическая доходность

## Прореживание данных

Прореживание — важная операция в анализе финансовых временных рядов. Обычно происходит *децимация*, или уменьшение частоты выборки, когда, например, тиковые данные агрегируются в минутные интервалы либо временной ряд, содержащий суточные значения, обобщается в недельные или месячные наблюдения (рис. 8.4).

In [22]: `data.resample('1w', label='right').last().head()` ❶  
 Out[22]:

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY	\
<b>Date</b>							
<b>2010-01-03</b>	NaN	NaN	NaN	NaN	NaN	NaN	
<b>2010-01-10</b>	30.282827	30.66	20.83	133.52	174.31	114.57	

2010-01-17	29.418542	30.86	20.80	127.14	165.21	113.64
2010-01-24	28.249972	28.96	19.91	121.43	154.12	109.21
2010-01-31	27.437544	28.18	19.40	125.41	148.72	107.39

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2010-01-03	NaN	NaN	1.4323	1096.35	NaN	NaN
2010-01-10	1144.98	18.13	1.4412	1136.10	49.84	111.37
2010-01-17	1136.03	17.91	1.4382	1129.90	47.42	110.86
2010-01-24	1091.76	27.31	1.4137	1092.60	43.79	107.17
2010-01-31	1073.87	24.62	1.3862	1081.05	40.72	105.96

In [23]: data.resample('1m', label='right').last().head() ❷  
 Out[23]:

	AAPL.O	MSFT.O	INTC.O	AMZN.O	GS.N	SPY \
Date						
2010-01-31	27.437544	28.1800	19.40	125.41	148.72	107.3900
2010-02-28	29.231399	28.6700	20.53	118.40	156.35	110.7400
2010-03-31	33.571395	29.2875	22.29	135.77	170.63	117.0000
2010-04-30	37.298534	30.5350	22.84	137.10	145.20	118.8125
2010-05-31	36.697106	25.8000	21.42	125.46	144.26	109.3690

	.SPX	.VIX	EUR=	XAU=	GDX	GLD
Date						
2010-01-31	1073.87	24.62	1.3862	1081.05	40.72	105.960
2010-02-28	1104.49	19.50	1.3625	1116.10	43.89	109.430
2010-03-31	1169.43	17.59	1.3510	1112.80	44.41	108.950
2010-04-30	1186.69	22.05	1.3295	1178.25	50.51	115.360
2010-05-31	1089.41	32.07	1.2305	1215.71	49.86	118.881

In [24]: rets.cumsum().apply(np.exp).resample('1m',  
 label='right').last().plot(figsize=(10, 6)); ❸

- ❶ Данные EOD (суточная выборка) агрегируются в недельные...
- ❷ ...и месячные интервалы.
- ❸ Построение графиков накапливаемой логарифмической доходности: сначала вызывается метод `cumsum()`, затем к полученным результатам применяется метод `np.exp()`, после чего выполняется месячное прореживание.

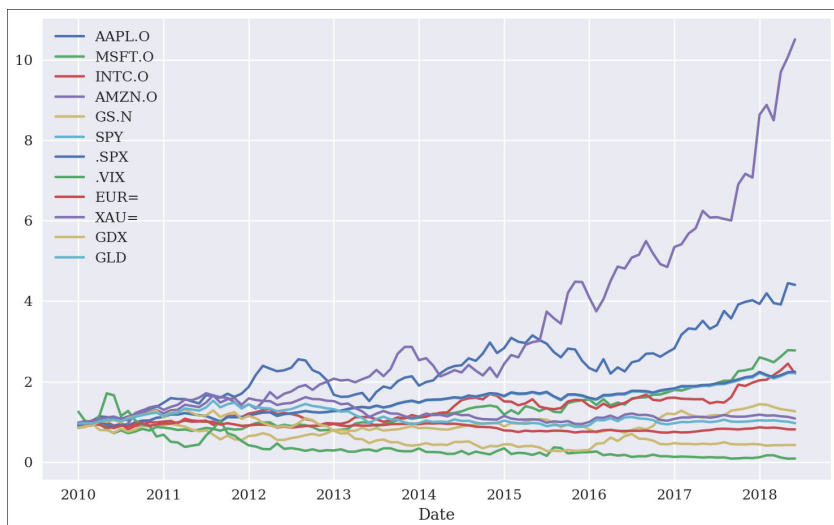


Рис. 8.4. Накапливаемая логарифмическая доходность  
(месячные интервалы)



### Избегайте смещенных прогнозов

Во многих операциях прореживания библиотека `pandas` по умолчанию использует метку `'left'` (выбор левого индекса). Но для получения корректных финансовых прогнозов следует использовать метку `'right'` (выбор правого индекса), т.е. самую последнюю доступную точку данных в интервале. В противном случае финансовый прогноз может оказаться смещенным<sup>3</sup>.

## Скользкая статистика

В финансовом анализе часто применяются *скользящие статистики*, называемые *индикаторами*. Они служат базовыми инструментами технического анализа. В этом разделе мы будем работать только с одним временным рядом из нашего набора.

<sup>3</sup> Смещенный прогноз возникает, когда в ходе финансового анализа используются данные, получаемые только в будущих периодах. Результат, как правило, оказывается “чересчур оптимистичным”. Это может проявляться, например, при тестировании торговой стратегии на исторических данных.

```
In [25]: sym = 'AAPL.O'
```

```
In [26]: data = pd.DataFrame(data[sym]).dropna()
```

```
In [27]: data.tail()
```

```
Out[27]:
```

AAPL.O	
Date	
2018-06-25	182.17
2018-06-26	184.43
2018-06-27	184.16
2018-06-28	185.50
2018-06-29	185.11

## Общие сведения

С помощью библиотеки `pandas` можно легко вычислить стандартные скользящие статистики.

```
In [28]: window = 20 ❶
```

```
In [29]: data['min'] = data[sym].rolling(window=window).min() ❷
```

```
In [30]: data['mean'] = data[sym].rolling(window=window).mean() ❸
```

```
In [31]: data['std'] = data[sym].rolling(window=window).std() ❹
```

```
In [32]: data['median'] = data[sym].rolling(window=window).median() ❺
```

```
In [33]: data['max'] = data[sym].rolling(window=window).max() ❻
```

```
In [34]: data['ewma'] = data[sym].ewm(halflife=0.5, \
                                     min_periods=window).mean() ❼
```

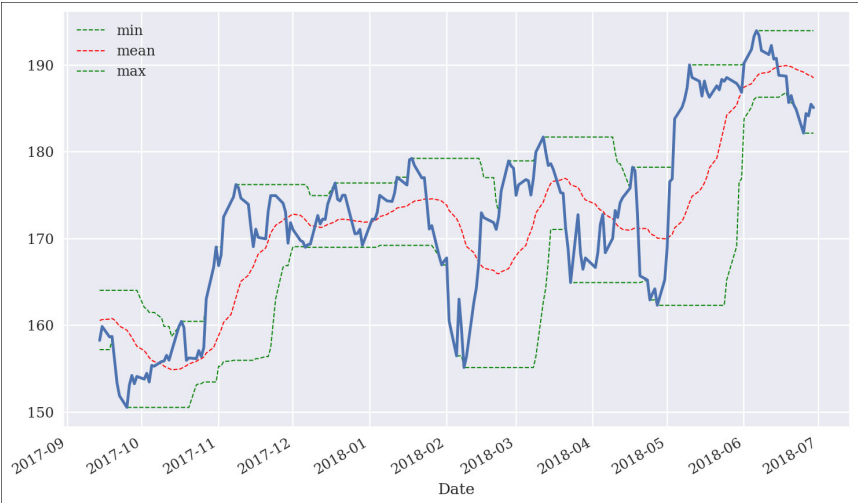
- ❶ Определение временного окна, т.е. числа охватываемых индексных значений.
- ❷ Вычисление скользящего минимального значения.
- ❸ Вычисление скользящего среднего.
- ❹ Вычисление скользящего среднеквадратического отклонения.
- ❺ Вычисление скользящей медианы.



- ⑥ Вычисление скользящего максимального значения.
- ⑦ Вычисление экспоненциально взвешенного скользящего среднего с полу-периодом затухания 0,5.

Для вычисления других технических индикаторов следует воспользоваться другими библиотеками (в частности, пакетом Cufflinks, описанным в главе 7). Можно также применять пользовательские индикаторы с помощью метода `apply()`.

В следующем примере выводится фрагмент имеющегося набора и визуализируется несколько скользящих статистик (рис. 8.5).



*Рис. 8.5. Скользящая статистика для минимального, среднего и максимального значений*

```
In [35]: data.dropna().head()
Out[35]:
```

	AAPL.O	min	mean	std \
Date				
2010-02-01	27.818544	27.437544	29.580892	0.933650
2010-02-02	27.979972	27.437544	29.451249	0.968048
2010-02-03	28.461400	27.437544	29.343035	0.950665
2010-02-04	27.435687	27.435687	29.207892	1.021129
2010-02-05	27.922829	27.435687	29.099892	1.037811

	median	max	ewma
Date			
2010-02-01	29.821542	30.719969	27.805432
2010-02-02	29.711113	30.719969	27.936337
2010-02-03	29.685970	30.719969	28.330134
2010-02-04	29.547113	30.719969	27.659299
2010-02-05	29.419256	30.719969	27.856947

```
In [36]: ax = data[['min', 'mean', 'max']].iloc[-200:].plot(
          figsize=(10, 6), style=['g--', 'r--', 'g--'], lw=0.8) ❶
          data[sym].iloc[-200:].plot(ax=ax, lw=2.0); ❷
```

- ❶ Построение графиков трех скользящих статистик для последних 200 строк набора.
- ❷ Добавление исходного временного ряда на диаграмму.

## Пример технического анализа

Скользящие статистические показатели — основной инструмент *технического анализа*, противопоставленного фундаментальному анализу, который в первую очередь опирается на данные финансовых отчетов и производственные показатели компании.

Традиционная торговая стратегия, основанная на техническом анализе, предполагает вычисление двух *простых скользящих средних* (Simple Moving Average — SMA). Идея заключается в том, что трейдер покупает акции (или другой финансовый инструмент), когда график краткосрочного SMA пересекает график долгосрочного SMA снизу вверх, и продает их, когда имеет место противоположная ситуация (пересечение сверху вниз). Реализовать такую стратегию можно с помощью объекта `DataFrame` библиотеки `pandas`.

Расчет скользящей статистики становится возможным только для наборов данных, размер которых больше временного окна (значения параметра `window`). На рис. 8.6 можно увидеть, что вычисление скользящего среднего начинается только тогда, когда набирается достаточное количество наблюдений с учетом параметров конкретного примера.

```
In [37]: data['SMA1'] = data[sym].rolling(window=42).mean() ❶

In [38]: data['SMA2'] = data[sym].rolling(window=252).mean() ❷

In [39]: data[['sym', 'SMA1', 'SMA2']].tail()
```

Out[39]:

	AAPL.O	SMA1	SMA2
Date			
2018-06-25	182.17	185.606190	168.265556
2018-06-26	184.43	186.087381	168.418770
2018-06-27	184.16	186.607381	168.579206
2018-06-28	185.50	187.089286	168.736627
2018-06-29	185.11	187.470476	168.901032

In [40]: data[[sym, 'SMA1', 'SMA2']].plot(figsize=(10, 6)); ❸

- ❶ Вычисление значений краткосрочного SMA.
- ❷ Вычисление значений долгосрочного SMA.
- ❸ Визуализация котировок и временных рядов SMA.



*Рис. 8.6. Котировки акций компании Apple и два простых скользящих средних*

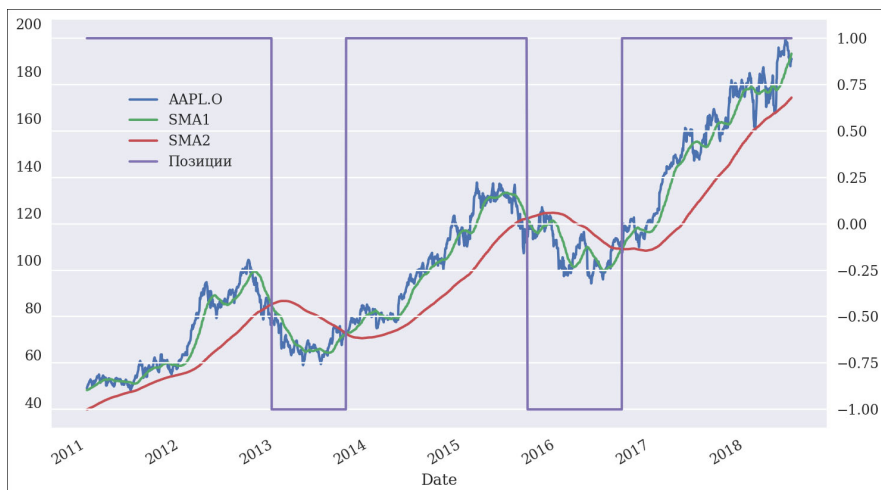
В рассматриваемом контексте SMA — просто удобный визуальный индикатор, задача которого — обозначить позиции для реализации торговой стратегии. На рис. 8.7 длиной позиции соответствует значение 1, а короткой позиции — значение -1. Изменение позиции обозначается (визуально) пересечением графиков двух SMA.

```
In [41]: data.dropna(inplace=True) ❶
```

```
In [42]: data['Позиции'] = np.where(data['SMA1'] > data['SMA2'], ❷  
                                     1, ❸  
                                     -1) ❹
```

```
In [43]: ax = data[['sym', 'SMA1', 'SMA2',  
                    'Позиции']].plot(figsize=(10, 6),  
                    secondary_y='Позиции', mark_right=False)  
ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

- ❶ Учитываются только полные строки данных.
- ❷ Если краткосрочное SMA больше долгосрочного SMA...
- ❸ ...то нужно открывать длинную позицию (уровень 1).
- ❹ В противном случае следует открывать короткую позицию (уровень -1).



*Рис. 8.7. Котировки акций компании Apple, два простых скользящих средних и позиции*

Неявная торговая стратегия приведет в данном случае к заключению всего нескольких сделок, которые будут происходить при изменении позиции (т.е. в точках пересечения графиков SMA). Учитывая сделки при открытии и закрытии торгов, суммарно будет всего шесть биржевых операций.

# Корреляционный анализ

Остальные возможности библиотеки `pandas` по обработке финансовых временных рядов мы будем рассматривать на примере фондового индекса S&P 500 и индекса волатильности VIX. Известно, что, когда индекс S&P 500 растет, индекс VIX обычно падает, и наоборот. Такое поведение индексов обусловлено корреляцией, а не причинно-следственными связями. В этом разделе мы постараемся найти статистическое подтверждение сильной отрицательной корреляции между индексами S&P 500 и VIX<sup>4</sup>.

## Исходные данные

Набор анализируемых данных теперь представлен двумя временными рядами, графики которых показаны на рис. 8.8.

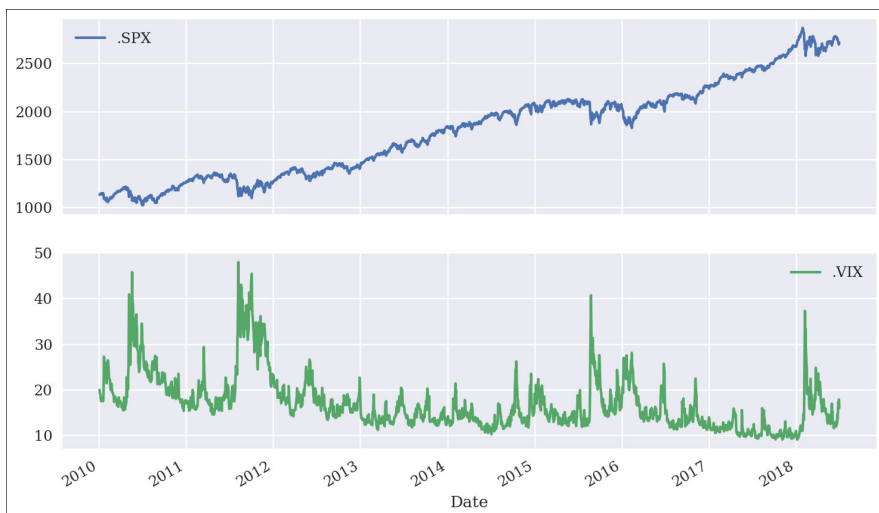


Рис. 8.8. Графики временных рядов для индексов S&P 500 и VIX (разные области построения)

```
In [44]: raw = pd.read_csv('../source/tr_eikon_eod_data.csv',  
                           index_col=0, parse_dates=True) ❶
```

```
In [45]: data = raw[['.SPX', '.VIX']].dropna()
```

<sup>4</sup> Одна из причин этого заключается в том, что, когда котировки падают (например, во время кризиса), объем торгов растет, а заодно повышается и волатильность. Когда котировки растут, инвесторы сохраняют спокойствие и не проявляют повышенной активности, пытаясь извлечь максимум из фазы роста.

```
In [46]: data.tail()
```

```
Out[46]:
```

	.SPX	.VIX
Date		
2018-06-25	2717.07	17.33
2018-06-26	2723.06	15.92
2018-06-27	2699.63	17.91
2018-06-28	2716.31	16.85
2018-06-29	2718.37	16.09

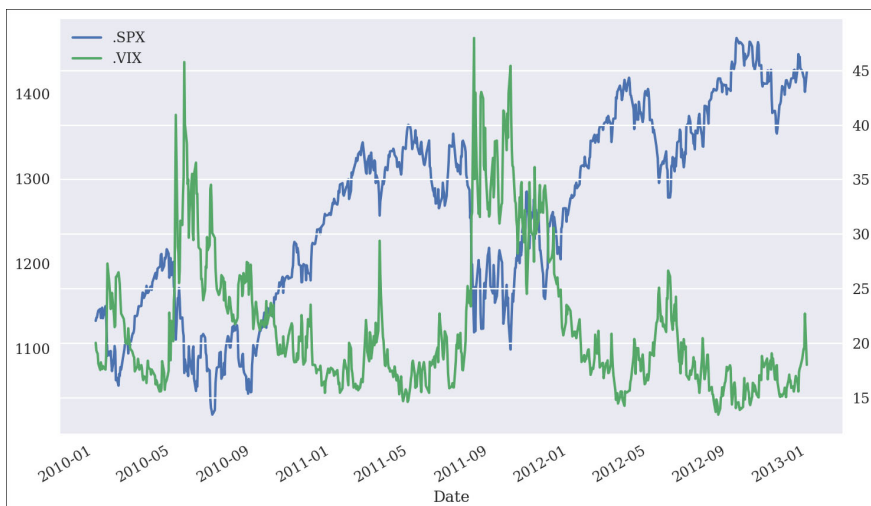
```
In [47]: data.plot(subplots=True, figsize=(10, 6));
```

- 1 Считывание данных EOD (получены через программный интерфейс Thomson Reuters Eikon Data) из CSV-файла.

Если наложить фрагменты графиков друг на друга, соответствующим образом настроив их масштаб, то факт существования отрицательной корреляции между индексами становится очевидным (рис. 8.9).

```
In [48]: data.loc['2012-12-31'].plot(secondary_y='.VIX',  
                                     figsize=(10, 6),  
                                     mark_right=False); 1
```

- 1 Вызов `.loc[:ДАТА]` позволяет получить выборку вплоть до указанной даты.



**Рис. 8.9.** Графики временных рядов для индексов S&P 500 и VIX (совмещенная диаграмма)

## Логарифмическая доходность

Как отмечалось выше, в статистическом анализе оперируют доходностями, а не изменениями абсолютных значений. Поэтому мы в первую очередь вычислим логарифмическую доходность, прежде чем анализировать другие показатели. На рис. 8.10 показаны графики высокочастотных изменений доходности. На обоих графиках можно заметить так называемые “кластеры волатильности”. Общая тенденция такова, что заметные колебания биржевых котировок сопровождаются синхронными изменениями индекса волатильности.

```
In [49]: rets = np.log(data / data.shift(1))
```

```
In [50]: rets.head()
```

```
Out[50]:
```

	.SPX	.VIX
Date		
2010-01-04	NaN	NaN
2010-01-05	0.003111	-0.035038
2010-01-06	0.000545	-0.009868
2010-01-07	0.003993	-0.005233
2010-01-08	0.002878	-0.050024

```
In [51]: rets.dropna(inplace=True)
```

```
In [52]: rets.plot(subplots=True, figsize=(10, 6));
```

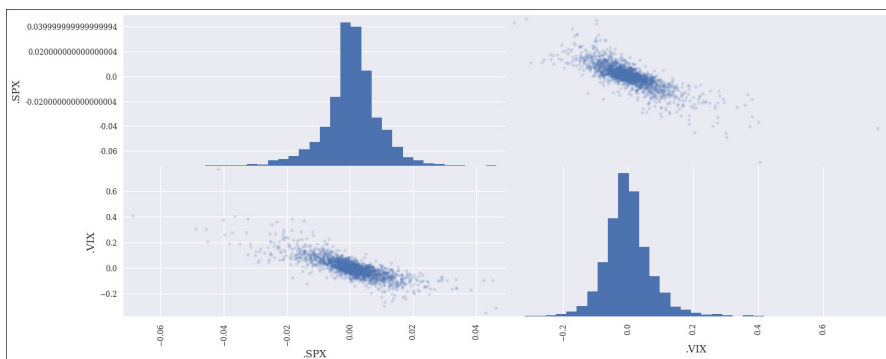


Рис. 8.10. Изменение логарифмической доходности индексов S&P 500 и VIX

Для визуализации подобных взаимосвязей в библиотеке `pandas` имеется функция `scatter_matrix()`. С ее помощью можно построить диаграмму зависимости двух логарифмических доходностей, снабдив ее гистограммой столбцов данных либо диаграммой ядерной оценки плотности (Kernel Density Estimator — KDE), как показано на рис. 8.11.

```
In [53]: pd.plotting.scatter_matrix(rets, ❶
                                     alpha=0.2, ❷
                                     diagonal='hist', ❸
                                     hist_kwds={'bins': 35}, ❹
                                     figsize=(15, 6));
```

- ❶ Визуализируемые данные.
- ❷ Параметр `alpha`, определяющий прозрачность точек.
- ❸ Вспомогательная диаграмма, выводимая по диагонали; в данном случае это гистограмма столбцов данных.
- ❹ Аргументы, передаваемые функции, которая отвечает за построение гистограммы.



**Рис. 8.11.** Диаграмма рассеяния логарифмической доходности индексов S&P 500 и VIX

## Регрессионный анализ по методу наименьших квадратов

Теперь можно легко выполнить регрессионный анализ по методу наименьших квадратов (МНК). На рис. 8.12 показана диаграмма рассеяния логарифмической доходности двух индексов, на которой через облако точек проходит

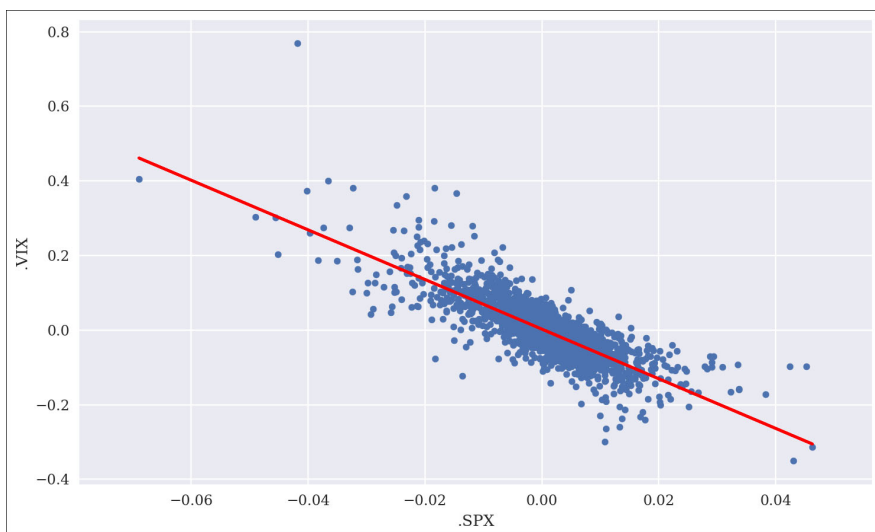


линия регрессии. Отрицательный наклон регрессионной прямой указывает на существование отрицательной корреляции между индексами.

```
In [54]: reg = np.polyfit(rets['.SPX'], rets['.VIX'], deg=1) ❶
```

```
In [55]: ax = rets.plot(kind='scatter', x='.SPX', y='.VIX',  
                        figsize=(10, 6)) ❷  
        ax.plot(rets['.SPX'], np.polyval(reg, rets['.SPX']), 'r',  
                lw=2); ❸
```

- ❶ Реализация линейной регрессии по методу наименьших квадратов.
- ❷ Построение диаграммы рассеяния логарифмической доходности...
- ❸ ...на которую накладывается линия регрессии.



*Рис. 8.12. Диаграмма рассеяния логарифмической доходности индексов S&P 500 и VIX с линией регрессии*

## Корреляция

Наконец, можно переходить к непосредственной оценке корреляции. Мы определим две метрики: статическую, рассчитанную для всего набора, и скользящую, вычисляемую в пределах временного окна. Как показано на рис. 8.13, корреляция действительно изменяется во времени, но при этом, учитывая параметризацию модели, она всегда отрицательная. Это служит наглядным

подтверждением того эмпирического факта, что между индексами S&P 500 и VIX существует (сильная) отрицательная корреляция.

```
In [56]: rets.corr() ❶
```

```
Out[56]:      .SPX      .VIX
.SPX    1.000000  -0.804382
.VIX   -0.804382   1.000000
```

```
In [57]: ax = rets['.SPX'].rolling(window=252).corr(
        rets['.VIX']).plot(figsize=(10, 6)) ❷
ax.axhline(rets.corr().iloc[0, 1], c='r'); ❸
```

- ❶ Корреляционная матрица для всего объекта DataFrame.
- ❷ График скользящей корреляции.
- ❸ Добавление статического значения в виде горизонтальной линии.



Рис. 8.13. Корреляция индексов S&P 500 и VIX (скользящая и статическая)

## Высокочастотные данные

В этой главе мы работаем с финансовыми временными рядами с помощью инструментов библиотеки `pandas`. Тиковые данные представляют собой частный случай таких рядов. Их можно трактовать примерно так же, как и данные EOD (котировки на конец торгов), с которыми мы имели дело в предыдущих

примерах. Импорт тиковых данных с помощью библиотеки `pandas` выполняется достаточно быстро. Вот как эта задача реализуется для набора из нескольких сот тысяч строк (рис. 8.14).

```
In [59]: %%time
# Источник данных: FXCM Forex Capital Markets Ltd.
tick = pd.read_csv('../source/fxcm_eur_usd_tick_data.csv',
                  index_col=0, parse_dates=True)
CPU times: user 1.07 s, sys: 149 ms, total: 1.22 s
Wall time: 1.16 s
```

```
In [60]: tick.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 461357 entries, 2018-06-29 00:00:00.082000 to
                2018-06-29 20:59:00.607000
Data columns (total 2 columns):
Bid    461357 non-null float64
Ask    461357 non-null float64
dtypes: float64(2)
memory usage: 10.6 MB
```

```
In [61]: tick['Mid'] = tick.mean(axis=1) ❶
```

```
In [62]: tick['Mid'].plot(figsize=(10, 6));
```

❶ Вычисление средней (Mid) цены для каждой строки.

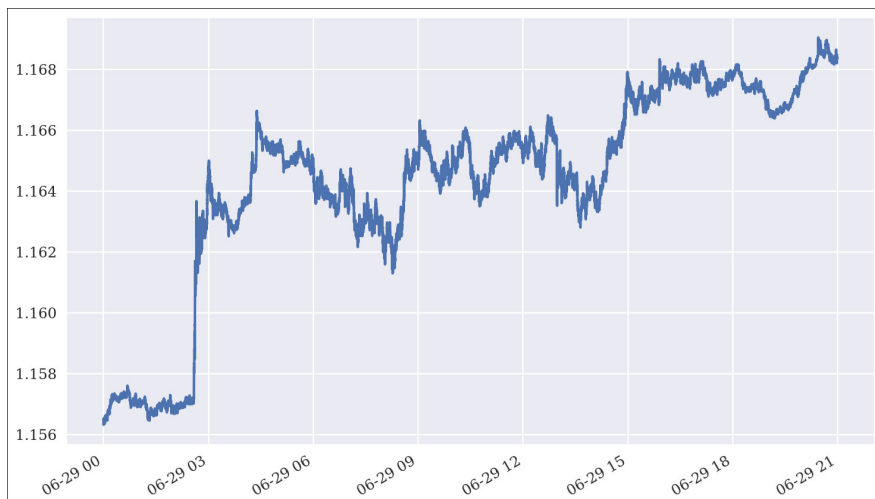


Рис. 8.14. Тиковые данные валютного курса EUR/USD

При работе с тиковыми данными часто приходится выполнять прореживание временного ряда. В следующем примере набор агрегируется в пятиминутные интервалы (рис. 8.15), что позволяет тестировать торговые стратегии на исторических данных или выполнять технический анализ.

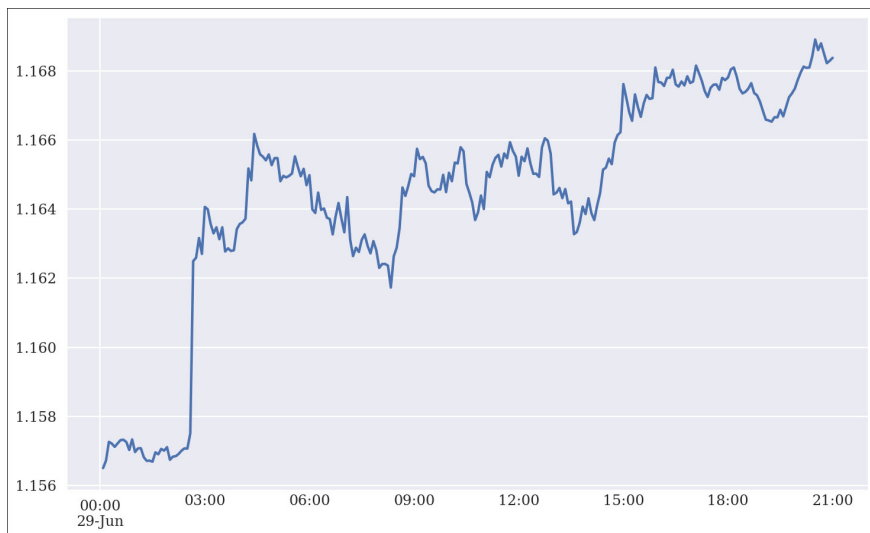
```
In [63]: tick_resam = tick.resample(rule='5min', label='right').last()
```

```
In [64]: tick_resam.head()
```

```
Out[64]:
```

	Bid	Ask	Mid
2018-06-29 00:05:00	1.15649	1.15651	1.156500
2018-06-29 00:10:00	1.15671	1.15672	1.156715
2018-06-29 00:15:00	1.15725	1.15727	1.157260
2018-06-29 00:20:00	1.15720	1.15722	1.157210
2018-06-29 00:25:00	1.15711	1.15712	1.157115

```
In [65]: tick_resam['Mid'].plot(figsize=(10, 6));
```



*Рис. 8.15. Данные валютного курса EUR/USD с пятиминутной агрегацией*

## Резюме

В этой главе были рассмотрены принципы обработки временных рядов — наиболее важного типа данных в финансовых вычислениях. Библиотека `pandas` содержит все необходимые средства не только для анализа таких данных, но и для их визуализации. Кроме того, библиотека позволяет считывать финансовые временные ряды из самых разных источников, а также экспортировать их в различные форматы, о чем будет рассказано в следующей главе.

## Дополнительные ресурсы

Хорошими источниками информации по теме данной главы послужат следующие книги.

- McKinney, Wes. *Python for Data Analysis* (2017, O'Reilly).
- VanderPlas, Jake. *Python Data Science Handbook* (2016, O'Reilly).

---

## Операции ввода-вывода

Создавать же версию, не имея фактов, — большая ошибка.

*Шерлок Холмс*

Как правило, большая часть данных — не только финансовых — хранится на жестких дисках (HDD) или запоминающих устройствах другого типа. Емкость устройств хранения все время растет, а их удельная стоимость (в пересчете на мегабайт данных) непрерывно снижается.

Фактически емкость хранилищ растет более высокими темпами, чем объемы оперативной памяти, доступной даже в самых мощных компьютерах. Поэтому возникает потребность не только помещать данные на диск для постоянного хранения, но и компенсировать нехватку оперативной памяти, создавая для нее файл подкачки на диске.

С учетом вышесказанного в финансовых и других приложениях, работающих с большими объемами данных, операции ввода-вывода играют важную роль. Зачастую именно они становятся узким местом, когда речь заходит о производительных вычислениях, поскольку скорость дискового обмена сильно ограничена<sup>1</sup>. Зачастую центральный процессор вынужден “простаивать” во время медленных операций ввода-вывода.

Несмотря на то что в большинстве финансовых и корпоративных систем приходится иметь дело с большими данными (где емкость уже измеряется петабайтами), отдельные аналитические задачи решаются на выборках среднего размера. Вот цитата из исследования, проведенного компанией Microsoft.

Собранные нами и другими исследователями данные свидетельствуют о том, что в большинстве реальных аналитических задач обрабатывается не более 100 Гбайт входных данных. В то же время такие популярные инфраструктуры, как Hadoop/MapReduce, изначально проектировались в расчете на петабайтовые объемы данных.

*Аппусвами (2013)*

---

<sup>1</sup> В данном случае не проводится никаких различий между разными уровнями оперативной памяти и кеша процессора. Оптимальное использование оперативной памяти — это отдельная тема.

Чаще всего в задачах финансового анализа обрабатывается максимум несколько гигабайтов данных — это именно тот объем, на который ориентированы встроенные инструменты Python и библиотеки научного стека, такие как NumPy, pandas и PyTables. Наборы данных такого размера можно обрабатывать в оперативной памяти, что с учетом вычислительных мощностей современных процессоров приводит к существенному повышению производительности. Но данные еще нужно прочитать с диска, а по окончании обработки результаты тоже записываются на диск, и это следует учитывать при рассмотрении вопросов производительности.

В главе рассматриваются следующие темы.

#### *Базовые операции ввода-вывода в Python*

В Python имеются встроенные функции сериализации и записи объектов на диск, а также считывания данных с диска в оперативную память. Помимо этого, Python располагает средствами работы с текстовыми файлами и реляционными базами данных. Библиотека NumPy также содержит специализированные функции для быстрого сохранения бинарных данных и чтения объектов ndarray.

#### *Ввод и вывод данных с помощью библиотеки pandas*

Библиотека pandas содержит множество удобных функций и методов, предназначенных для чтения и записи данных в различных файловых форматах (включая CSV и JSON).

#### *Ввод и вывод данных с помощью PyTables*

PyTables — это пакет для управления иерархическими базами данных, основанный на стандарте HDF5 и ориентированный на выполнение быстрых операций ввода-вывода с огромными наборами данных. Скорость в данном случае ограничена зачастую только техническими возможностями используемого оборудования.

#### *Ввод и вывод данных с помощью TsTables*

TsTables — это надстройка к пакету PyTables, ускоряющая чтение и запись временных рядов.

## **Базовые операции ввода-вывода в Python**

В Python имеется множество средств ввода-вывода, часть из которых оптимизирована с учетом производительности, а часть — с учетом гибкости использования. Но в целом со всеми этими инструментами удобно работать как в интерактивном режиме, так и в производственной среде.

## Запись объектов на диск

Иногда объекты Python требуется записывать на диск, например для использования в других проектах, передачи другим разработчикам или написания документации к имеющемуся коду. Для решения такого рода задач предназначен модуль `pickle`, позволяющий сериализовать большинство объектов Python. *Сериализация* означает преобразование объекта из иерархической структуры в байтовый поток. Обратная операция называется *десериализацией*.

Как всегда, сначала нужно импортировать рабочие модули и задать ряд базовых настроек (для последующего построения диаграмм).

```
In [1]: from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

В следующем примере мы будем работать с псевдослучайными данными, которые хранятся в объекте `list` (список).

```
In [2]: import pickle ❶
        import numpy as np
        from random import gauss ❷

In [3]: a = [gauss(1.5, 2) for i in range(1000000)] ❸

In [4]: path = '/Users/yves/Temp/data/' ❹

In [5]: pkl_file = open(path + 'data.pkl', 'wb') ❺
```

- ❶ Импорт модуля `pickle` из стандартной библиотеки.
- ❷ Импорт модуля `gauss` для генерирования случайных чисел с нормальным распределением.
- ❸ Создание большого списка, заполняемого случайными числами.
- ❹ Путь для хранения файлов данных<sup>2</sup>.
- ❺ Открытие файла для записи в двоичном режиме.

---

<sup>2</sup> Можете, например, создать подкаталог `temp` в текущем каталоге и указать путь `'./temp/'`. — Примеч. ред.



За сериализацию и десериализацию объектов Python отвечают функции `pickle.dump()` и `pickle.load()`.

```
In [6]: %time pickle.dump(a, pkl_file) ❶  
CPU times: user 37.2 ms, sys: 15.3 ms, total: 52.5 ms  
Wall time: 50.8 ms
```

```
In [7]: pkl_file.close() ❷
```

```
In [8]: ll $path* ❸  
-rw-r--r--  1 yves  staff  9002006 Oct 19 12:11  
/Users/yves/Temp/data/data.pkl
```

```
In [9]: pkl_file = open(path + 'data.pkl', 'rb') ❹
```

```
In [10]: %time b = pickle.load(pkl_file) ❺  
CPU times: user 34.1 ms, sys: 16.7 ms, total: 50.8 ms  
Wall time: 48.7 ms
```

```
In [11]: a[:3]  
Out[11]: [6.517874180585469, -0.5552400459507827, 2.8488946310833096]
```

```
In [12]: b[:3]  
Out[12]: [6.517874180585469, -0.5552400459507827, 2.8488946310833096]
```

```
In [13]: np.allclose(np.array(a), np.array(b)) ❻  
Out[13]: True
```

- ❶ Сериализация и сохранение объекта `a` в файл.
- ❷ Заккрытие файла.
- ❸ Проверяем наличие файла на диске и его размер (Mac/Linux).
- ❹ Открытие файла для чтения в двоичном режиме (`rb`).
- ❺ Чтение объекта с диска и его десериализация.
- ❻ Преобразование объектов `a` и `b` в объекты `ndarray`. Функция `np.allclose()` позволяет убедиться в том, что оба объекта хранят одинаковые данные (числа).

Как видите, сохранить и загрузить отдельный объект с помощью модуля `pickle` совсем не сложно. А как насчет сразу двух объектов?

```
In [14]: pkl_file = open(path + 'data.pkl', 'wb')
```

```
In [15]: %time pickle.dump(np.array(a), pkl_file) ❶  
CPU times: user 58.1 ms, sys: 6.09 ms, total: 64.2 ms  
Wall time: 32.5 ms
```

```
In [16]: %time pickle.dump(np.array(a) ** 2, pkl_file) ❷  
CPU times: user 66.7 ms, sys: 7.22 ms, total: 73.9 ms  
Wall time: 39.3 ms
```

```
In [17]: pkl_file.close()
```

```
In [18]: ll $path* ❸  
-rw-r--r-- 1 yves staff 16000322 Oct 19 12:11  
/Users/yves/Temp/data/data.pkl
```

- ❶ Сериализация и сохранение `ndarray`-версии объекта `a`.
- ❷ Сериализация и сохранение `ndarray`-версии объекта `a` с возведением в квадрат.
- ❸ Размер файла увеличился почти вдвое.

Теперь рассмотрим, как обратно загрузить два объекта `ndarray`.

```
In [19]: pkl_file = open(path + 'data.pkl', 'rb')
```

```
In [20]: x = pickle.load(pkl_file) ❶  
x[:4]  
Out[20]: array([ 6.51787418, -0.55524005,  2.84889463,  5.94489175 ])
```

```
In [21]: y = pickle.load(pkl_file) ❷  
y[:4]  
Out[21]: array([42.48268383,  0.30829151,  8.11620062, 35.34173791])
```

```
In [22]: pkl_file.close()
```

- ❶ Получение объекта, сохраненного *первым*.
- ❷ Получение объекта, сохраненного *вторым*.

Модуль `pickle` сохраняет объекты по принципу FIFO (First In, First Out — первым пришел, первым ушел). Проблема здесь только одна: отсутствие метаданных, с помощью которой пользователь мог бы заранее узнать, что хранится в сериализованном файле.

Одно из решений заключается в том, чтобы записывать в файл не одиночные объекты, а словарь, включающий все объекты.

```
In [23]: pickle_file = open(path + 'data.pkl', 'wb')
         pickle.dump({'x': x, 'y': y}, pickle_file) ❶
         pickle_file.close()

In [24]: pickle_file = open(path + 'data.pkl', 'rb')
         data = pickle.load(pickle_file) ❷
         pickle_file.close()
         for key in data.keys():
             print(key, data[key][:4])
         x [ 6.51787418 -0.55524005  2.84889463  5.94489175]
         y [42.48268383  0.30829151  8.11620062 35.34173791]

In [25]: !rm -f $path*
```

❶ Сохранение словаря, содержащего два объекта `ndarray`.

❷ Считывание словаря.

В этом способе все объекты записываются и считываются одновременно, что в большинстве случаев можно считать разумным компромиссом.



### Проблемы совместимости

Сериализовывать объекты с помощью модуля `pickle` достаточно просто. Следует опасаться только одного: при обновлении пакета новая версия может не распознать сериализованные объекты, созданные предыдущей версией. Трудности могут возникнуть и при обмене объектами между разными операционными системами и платформами. Именно поэтому предпочтительнее использовать встроенные средства ввода-вывода таких библиотек, как `NumPy` и `pandas`, о чем мы поговорим в следующих разделах.

## Чтение и запись текстовых файлов

Python располагает богатым инструментарием для работы с текстовыми данными, что находит активное применение в корпоративных и научных приложениях. В Python можно работать как со строковыми объектами, так и с текстовыми файлами в целом.

Предположим, необходимо передать большой объем данных в виде CSV-файла. Несмотря на то что такие файлы имеют внутреннюю структуру, по сути это простые текстовые файлы. В следующем примере создаются два объекта:

ndarray, содержащий набор случайных чисел, и DatetimeIndex, содержащий почасовые данные. Затем они объединяются и сохраняются в CSV-файле.

```
In [26]: import pandas as pd
```

```
In [27]: rows = 5000 ❶  
         a = np.random.standard_normal((rows, 5)).round(4) ❷
```

```
In [28]: a ❷  
Out[28]: array([[ -0.0892, -1.0508, -0.5942,  0.3367,  1.508 ],  
                [  2.1046,  3.2623,  0.704 , -0.2651,  0.4461],  
                [-0.0482, -0.9221,  0.1332,  0.1192,  0.7782],  
                ...,  
                [  0.3026, -0.2005, -0.9947,  1.0203, -0.6578],  
                [-0.7031, -0.6989, -0.8031, -0.4271,  1.9963],  
                [  2.4573,  2.2151,  0.158 , -0.7039, -1.0337]])
```

```
In [29]: t = pd.date_range(start='2019/1/1', periods=rows,  
                           freq='H') ❸
```

```
In [30]: t ❸  
Out[30]: DatetimeIndex(['2019-01-01 00:00:00', '2019-01-01 01:00:00',  
                        '2019-01-01 02:00:00', '2019-01-01 03:00:00',  
                        '2019-01-01 04:00:00', '2019-01-01 05:00:00',  
                        '2019-01-01 06:00:00', '2019-01-01 07:00:00',  
                        '2019-01-01 08:00:00', '2019-01-01 09:00:00',  
                        ...,  
                        '2019-07-27 22:00:00', '2019-07-27 23:00:00',  
                        '2019-07-28 00:00:00', '2019-07-28 01:00:00',  
                        '2019-07-28 02:00:00', '2019-07-28 03:00:00',  
                        '2019-07-28 04:00:00', '2019-07-28 05:00:00',  
                        '2019-07-28 06:00:00', '2019-07-28 07:00:00'],  
                        dtype='datetime64[ns]', length=5000, freq='H')
```

```
In [31]: csv_file = open(path + 'data.csv', 'w') ❹
```

```
In [32]: header = 'date,no1,no2,no3,no4,no5\n' ❺
```

```
In [33]: csv_file.write(header) ❻  
Out[33]: 25
```

```
In [34]: for t_, (no1, no2, no3, no4, no5) in zip(t, a): ❼
```

```
s = '{}',{},{},{},{},{},{}\n'.format(t_, no1, no2, no3,  
no4, no5) ⑦  
csv_file.write(s) ⑧
```

```
In [35]: csv_file.close()
```

```
In [36]: ll $path*  
-rw-r--r-- 1 yves staff 284757 Oct 19 12:11  
/Users/yves/Temp/data/data.csv
```

- ① Количество строк в наборе данных.
- ② Создание объекта `ndarray`, наполняемого случайными числами.
- ③ Создание объекта `DatetimeIndex` необходимой длины (с часовым интервалом выборки).
- ④ Открытие файла для записи.
- ⑤ Создание строки заголовков (названия столбцов) и запись ее в качестве первой строки файла.
- ⑥ Построчное объединение двух массивов...
- ⑦ ...и получение объектов `str`...
- ⑧ ...которые последовательно записываются в CSV-файл (с добавлением в конец).

Обратная операция реализуется несложно. Сначала нужно открыть существующий CSV-файл, а затем построчно прочитать его содержимое с помощью метода `readline()` или `readlines()` объекта `file`.

```
In [37]: csv_file = open(path + 'data.csv', 'r') ①
```

```
In [38]: for i in range(5): ②  
    print(csv_file.readline(), end='')  
    date,no1,no2,no3,no4,no5  
2019-01-01 00:00:00,-0.0892,-1.0508,-0.5942,0.3367,1.508  
2019-01-01 01:00:00,2.1046,3.2623,0.704,-0.2651,0.4461  
2019-01-01 02:00:00,-0.0482,-0.9221,0.1332,0.1192,0.7782  
2019-01-01 03:00:00,-0.359,-2.4955,0.6164,0.712,-1.4328
```

```
In [39]: csv_file.close()
```

```
In [40]: csv_file = open(path + 'data.csv', 'r') ❶
```

```
In [41]: content = csv_file.readlines() ❸
```

```
In [42]: content[:5] ❹
```

```
Out[42]: ['date,no1,no2,no3,no4,no5\n',  
          '2019-01-01 00:00:00,-0.0892,-1.0508,-0.5942,0.3367,1.508\n',  
          '2019-01-01 01:00:00,2.1046,3.2623,0.704,-0.2651,0.4461\n',  
          '2019-01-01 02:00:00,-0.0482,-0.9221,0.1332,0.1192,0.7782\n',  
          '2019-01-01 03:00:00,-0.359,-2.4955,0.6164,0.712,-1.4328\n']
```

```
In [43]: csv_file.close()
```

- ❶ Открытие файла для чтения (r).
- ❷ Построчное считывание содержимого файла с выводом на экран.
- ❸ Считывание всего файла за один раз.
- ❹ В результате формируется список, включающий все строки файла в виде объектов str.

Файлы формата CSV настолько распространены, что в стандартную библиотеку Python включен специальный модуль `csv`, упрощающий работу с ними. Этот модуль содержит два объекта `reader` (итераторы), которые возвращают либо список списков, либо список словарей.

```
In [44]: import csv
```

```
In [45]: with open(path + 'data.csv', 'r') as f:  
          csv_reader = csv.reader(f) ❶  
          lines = [line for line in csv_reader]
```

```
In [46]: lines[:5] ❶
```

```
Out[46]: [['date', 'no1', 'no2', 'no3', 'no4', 'no5'],  
          ['2019-01-01 00:00:00', '-0.0892', '-1.0508', '-0.5942',  
           '0.3367', '1.508'],  
          ['2019-01-01 01:00:00', '2.1046', '3.2623', '0.704',  
           '-0.2651', '0.4461'],  
          ['2019-01-01 02:00:00', '-0.0482', '-0.9221', '0.1332',  
           '0.1192', '0.7782'],  
          ['2019-01-01 03:00:00', '-0.359', '-2.4955', '0.6164',  
           '0.712', '-1.4328']]
```

```
In [47]: with open(path + 'data.csv', 'r') as f:
        csv_reader = csv.DictReader(f) ❷
        lines = [line for line in csv_reader]

In [48]: lines[:3] ❷
Out[48]: [OrderedDict([('date', '2019-01-01 00:00:00'),
                        ('no1', '-0.0892'),
                        ('no2', '-1.0508'),
                        ('no3', '-0.5942'),
                        ('no4', '0.3367'),
                        ('no5', '1.508')]),
          OrderedDict([('date', '2019-01-01 01:00:00'),
                        ('no1', '2.1046'),
                        ('no2', '3.2623'),
                        ('no3', '0.704'),
                        ('no4', '-0.2651'),
                        ('no5', '0.4461')]),
          OrderedDict([('date', '2019-01-01 02:00:00'),
                        ('no1', '-0.0482'),
                        ('no2', '-0.9221'),
                        ('no3', '0.1332'),
                        ('no4', '0.1192'),
                        ('no5', '0.7782')])]
```

```
In [49]: !rm -f $path*
```

- ❶ Функция `csv.reader()` возвращает строки CSV-файла в виде списка.
- ❷ Функция `csv.DictReader()` возвращает строки CSV-файла в виде объектов `OrderedDict`, которые являются частным случаем словаря.

## Работа с реляционными базами данных

Python умеет работать с любыми реляционными СУБД и базами данных NoSQL. В состав Python по умолчанию включена СУБД SQLite3 ([www.sqlite.org](http://www.sqlite.org)). Мы воспользуемся ею, чтобы продемонстрировать принципы работы с реляционными базами данных<sup>3</sup>.

---

<sup>3</sup> Обзор интерфейсов доступа к базам данных в Python доступен по адресу <https://wiki.python.org/moin/DatabaseInterfaces>. Вместо того чтобы работать с реляционными базами данных напрямую, лучше применять ORM-библиотеки наподобие SQLAlchemy ([www.sqlalchemy.org](http://www.sqlalchemy.org)). Они добавляют уровень абстракции, позволяющий писать объектно-ориентированный код в стиле Python. Кроме того, они дают возможность переключаться на разные серверные СУБД путем изменения конфигурации.

```
In [50]: import sqlite3 as sq3
```

```
In [51]: con = sq3.connect(path + 'numbs.db') ❶
```

```
In [52]: query = 'CREATE TABLE numbs (Date date, No1 real, No2 real)' ❷
```

```
In [53]: con.execute(query) ❸
```

```
Out[53]: <sqlite3.Cursor at 0x102655f10>
```

```
In [54]: con.commit() ❹
```

```
In [55]: q = con.execute ❺
```

```
In [56]: q('SELECT * FROM sqlite_master').fetchall() ❻
```

```
Out[56]: [('table',  
          'numbs',  
          'numbs',  
          2,  
          'CREATE TABLE numbs (Date date, No1 real, No2 real)')]
```

- ❶ Настройка подключения к базе данных. Если файла не существует, он будет создан.
- ❷ SQL-запрос на создание таблицы с тремя столбцами<sup>4</sup>.
- ❸ Выполнение запроса...
- ❹ ...и фиксация изменений.
- ❺ Определение короткого псевдонима для метода `con.execute()`.
- ❻ Извлечение метаинформации о базе данных.

После установки подключения к базе данных и создания таблицы можно заполнить ее данными. Каждая строка таблицы будет содержать объект `datetime` и два объекта типа `float`.

```
In [57]: import datetime
```

```
In [58]: now = datetime.datetime.now()
```

```
q('INSERT INTO numbs VALUES(?, ?, ?)', (now, 0.12, 7.3)) ❶
```

```
Out[58]: <sqlite3.Cursor at 0x102655f80>
```

---

<sup>4</sup> Синтаксис языка SQLite3 детально описан на официальном сайте (<https://www.sqlite.org/lang.html>).



```

In [59]: np.random.seed(100)

In [60]: data = np.random.standard_normal((10000, 2)).round(4) ❷

In [61]: %%time
         for row in data: ❸
             now = datetime.datetime.now()
             q('INSERT INTO numbs VALUES(?, ?, ?)', \
               (now, row[0], row[1]))
         con.commit()
CPU times: user 115 ms, sys: 6.69 ms, total: 121 ms
Wall time: 124 ms

In [62]: q('SELECT * FROM numbs').fetchmany(4) ❹
Out[62]: [('2018-10-19 12:11:15.564019', 0.12, 7.3),
          ('2018-10-19 12:11:15.592956', -1.7498, 0.3427),
          ('2018-10-19 12:11:15.593033', 1.153, -0.2524),
          ('2018-10-19 12:11:15.593051', 0.9813, 0.5142)]

In [63]: q('SELECT * FROM numbs WHERE no1 > 0.5').fetchmany(4) ❺
Out[63]: [('2018-10-19 12:11:15.593033', 1.153, -0.2524),
          ('2018-10-19 12:11:15.593051', 0.9813, 0.5142),
          ('2018-10-19 12:11:15.593104', 0.6727, -0.1044),
          ('2018-10-19 12:11:15.593134', 1.619, 1.5416)]

In [64]: pointer = q('SELECT * FROM numbs') ❻

In [65]: for i in range(3):
         print(pointer.fetchone()) ❼
          ('2018-10-19 12:11:15.564019', 0.12, 7.3)
          ('2018-10-19 12:11:15.592956', -1.7498, 0.3427)
          ('2018-10-19 12:11:15.593033', 1.153, -0.2524)

In [66]: rows = pointer.fetchall() ❽
         rows[:3]
Out[66]: [('2018-10-19 12:11:15.593051', 0.9813, 0.5142),
          ('2018-10-19 12:11:15.593063', 0.2212, -1.07),
          ('2018-10-19 12:11:15.593073', -0.1895, 0.255)]

```

- ❶ Добавление строки в таблицу numbs.
- ❷ Создание объекта ndaггау, содержащего случайные числа.
- ❸ Прокрутка содержимого объекта ndaггау.

- ④ Извлечение нескольких строк из таблицы.
- ⑤ То же самое, но с условием для столбца no1.
- ⑥ Создание объекта-указателя...
- ⑦ ...выступающего в качестве генератора.
- ⑧ Извлечение всех остальных строк.

Если таблица больше не нужна, ее можно удалить из базы данных.

```
In [67]: q('DROP TABLE IF EXISTS numbs') ①
```

```
Out[67]: <sqlite3.Cursor at 0x1187a7420>
```

```
In [68]: q('SELECT * FROM sqlite_master').fetchall() ②
```

```
Out[68]: []
```

```
In [69]: con.close() ③
```

```
In [70]: !rm -f $path* ④
```

- ① Удаление таблицы из базы данных.
- ② Теперь в базе данных отсутствуют таблицы.
- ③ Закрытие подключения к базе данных.
- ④ Удаление файла базы данных с диска.

Реляционные базы данных — достаточно обширная тема, к тому же слишком сложная, чтобы полноценно рассмотреть ее здесь. Вам достаточно знать следующее:

- Python прекрасно работает практически с любыми базами данных;
- синтаксис SQL зависит от конкретной СУБД, а все остальное реализует-ся на уровне Python.

В последующих разделах будет приведен ряд других примеров работы с SQLite3.

## Считывание и запись массивов NumPy

Библиотека NumPy располагает удобными и производительными функциями для считывания и записи объектов `ndarray`. Это может пригодиться, например, когда необходимо преобразовать объекты NumPy типа `dtype` в один

из типов данных, специфичных для конкретной СУБД (такой, как SQLite3). Чтобы продемонстрировать эффективность NumPy, возьмем за основу пример из предыдущего раздела.

На этот раз, в отличие от библиотеки `pandas`, мы воспользуемся функцией `np.arange()`, чтобы сгенерировать объект `ndarray`, включающий объект `datetime`.

```
In [71]: dtimes = np.arange('2019-01-01 10:00:00', \
                             '2025-12-31 22:00:00', dtype='datetime64[m]') ❶
```

```
In [72]: len(dtimes)
Out[72]: 3681360
```

```
In [73]: dtype = np.dtype([('Date', 'datetime64[m]'),
                             ('No1', 'f'), ('No2', 'f')]) ❷
```

```
In [74]: data = np.zeros(len(dtimes), dtype=dtype) ❸
```

```
In [75]: data['Date'] = dtimes ❹
```

```
In [76]: a = np.random.standard_normal((len(dtimes), 2)).round(4) ❺
```

```
In [77]: data['No1'] = a[:, 0] ❻
          data['No2'] = a[:, 1] ❻
```

```
In [78]: data.nbytes ❼
Out[78]: 58901760
```

- ❶ Создание объекта `ndarray` для хранения данных типа `datetime`.
- ❷ Создание специального объекта структурированного массива.
- ❸ Создание объекта `ndarray` для хранения структурированного массива.
- ❹ Заполнение столбца `Date`.
- ❺ Генерирование случайных чисел...
- ❻ ...добавляемых в столбцы `No1` и `No2`.
- ❼ Размер структурированного массива в байтах.

Операции сохранения объектов `ndarray` оптимизированы и выполняются очень быстро. В данном случае массив размером почти 60 Мбайт сохраняется

за долю секунды (на SSD-диске). Для сохранения более крупного объекта `ndarray` размером 480 Мбайт потребуется примерно полсекунды<sup>5</sup>.

```
In [79]: %time np.save(path + 'array', data) ❶
CPU times: user 37.4 ms, sys: 58.9 ms, total: 96.4 ms
Wall time: 77.9 ms

In [80]: ll $path* ❷
-rw-r--r--  1 yves  staff  58901888 Oct 19 12:11
/Users/yves/Temp/data/array.npy

In [81]: %time np.load(path + 'array.npy') ❸
CPU times: user 1.67 ms, sys: 44.8 ms, total: 46.5 ms
Wall time: 44.6 ms
Out[81]: array([('2019-01-01T10:00',  1.5131,  0.6973),
                ('2019-01-01T10:01', -1.722 , -0.4815),
                ('2019-01-01T10:02',  0.8251,  0.3019), ...,
                ('2025-12-31T21:57',  1.372 ,  0.6446),
                ('2025-12-31T21:58', -1.2542,  0.1612),
                ('2025-12-31T21:59', -1.1997, -1.097 )],
               dtype=[('Date', '<M8[m]'), ('No1', '<f4'),
                       ('No2', '<f4')])

In [82]: %time data = np.random.standard_normal((10000,
                                                    6000)).round(4) ❹
CPU times: user 2.69 s, sys: 391 ms, total: 3.08 s
Wall time: 2.78 s

In [83]: data.nbytes ❺
Out[83]: 480000000

In [84]: %time np.save(path + 'array', data) ❻
CPU times: user 42.9 ms, sys: 300 ms, total: 343 ms
Wall time: 481 ms

In [85]: ll $path* ❼
-rw-r--r--  1 yves  staff  480000128 Oct 19 12:11
/Users/yves/Temp/data/array.npy
```

---

<sup>5</sup> Учтите, что при многократном повторении операции даже на одном компьютере время записи может существенно меняться, поскольку оно зависит от множества факторов, включая степень загрузки центрального процессора и каналов ввода-вывода.

```
In [86]: %time np.load(path + 'array.npy') ❹
          CPU times: user 2.32 ms, sys: 363 ms, total: 365 ms
          Wall time: 363 ms
Out[86]: array([[ 0.3066,  0.5951,  0.5826, ...,  1.6773,
                  0.4294, -0.2216],
                [ 0.8769,  0.7292, -0.9557, ...,  0.5084,
                  0.9635, -0.4443],
                [-1.2202, -2.5509, -0.0575, ..., -1.6128,
                  0.4662, -1.3645],
                ...,
                [-0.5598,  0.2393, -2.3716, ...,  1.7669,
                  0.2462,  1.035 ],
                [ 0.273 ,  0.8216, -0.0749, ..., -0.0552,
                 -0.8396,  0.3077],
                [-0.6305,  0.8331,  1.3702, ...,  0.3493,
                  0.1981,  0.2037]])
```

```
In [87]: !rm -f $path*
```

- ❶ Сохранение структурированного объекта `ndarray` на диск.
- ❷ Объект занимает на диске примерно столько же места, сколько и в памяти (благодаря сохранению в бинарном формате).
- ❸ Загрузка структурированного объекта `ndarray` с диска.
- ❹ Создание более крупного объекта `ndarray`.

Эти примеры наглядно демонстрируют, что скорость записи в основном определяется производительностью дисковой подсистемы. В данном случае мы получили скорость, характерную для типичного SSD-накопителя (порядка 500 Мбайт/с).

В любом случае такие операции чтения и записи работают быстрее, чем функции реляционных баз данных или функции сериализации модуля `pickle`. На то есть две причины: во-первых, наши данные имеют числовой тип, а во-вторых, в NumPy применяется бинарный формат хранения, что сводит накладные расходы практически к нулю. Конечно, такой подход лишает нас многих функциональных возможностей, свойственных реляционным базам данных, но это можно компенсировать с помощью пакета `PyTables`, который будет рассматриваться в последующих разделах.

# Ввод и вывод данных с помощью библиотеки pandas

Одно из основных преимуществ библиотеки pandas заключается в наличии встроенной поддержки различных форматов данных, включая:

- CSV (значения, разделенные запятыми);
- SQL (Structured Query Language);
- XLS/XLSX (файлы Microsoft Excel);
- JSON (JavaScript Object Notation);
- HTML (HyperText Markup Language).

В табл. 9.1 перечислены все поддерживаемые библиотекой pandas форматы данных, а также соответствующие методы импорта и экспорта, доступные в классе DataFrame. Параметры таких методов описаны в документации (например, описание метода `pd.read_csv()` доступно по адресу <http://bit.ly/2DaB9C7>).

Таблица 9.1. Методы импорта и экспорта данных

Формат	Импорт	Экспорт	Примечание
CSV	<code>pd.read_csv()</code>	<code>.to_csv()</code>	Текстовый файл
XLS/XLSX	<code>pd.read_excel()</code>	<code>.to_excel()</code>	Электронная таблица
HDF	<code>pd.read_hdf()</code>	<code>.to_hdf()</code>	База данных HDF5
SQL	<code>pd.read_sql()</code>	<code>.to_sql()</code>	Таблица SQL
JSON	<code>pd.read_json()</code>	<code>.to_json()</code>	Объект JSON
MSGPACK	<code>pd.read_msgpack()</code>	<code>.to_msgpack()</code>	Двоичный формат обмена данными
HTML	<code>pd.read_html()</code>	<code>.to_html()</code>	HTML-разметка
GBQ	<code>pd.read_gbq()</code>	<code>.to_gbq()</code>	Формат Big Query, разработанный компанией Google
DTA	<code>pd.read_stata()</code>	<code>.to_stata()</code>	Форматы 104, 105, 108, 113–115 и 117
Любой	<code>pd.read_clipboard()</code>	<code>.to_clipboard()</code>	В том числе документы HTML
Любой	<code>pd.read_pickle()</code>	<code>.to_pickle()</code>	Объект Python (в том числе структурированный)

Для тестирования возможностей библиотеки pandas мы сформируем большой массив вещественных чисел.

```
In [88]: data = np.random.standard_normal((1000000, 5)).round(4)
```

```
In [89]: data[:3]
Out[89]: array([[ 0.4918,  1.3707,  0.137 ,  0.3981, -1.0059],
                [ 0.4516,  1.4445,  0.0555, -0.0397,  0.44  ],
                [ 0.1629, -0.8473, -0.8223, -0.4621, -0.5137]])
```

Мы также вернемся к рассмотрению базы данных SQLite3 и сравним ее производительность с производительностью библиотеки `pandas`.

## Работа с реляционными базами данных

Следующий код SQLite3 уже должен быть вам понятен.

```
In [90]: filename = path + 'numbers'

In [91]: con = sq3.Connection(filename + '.db')

In [92]: query = 'CREATE TABLE numbers (No1 real, No2 real,\
                No3 real, No4 real, No5 real)' ❶

In [93]: q = con.execute
         qm = con.executemany

In [94]: q(query)
Out[94]: <sqlite3.Cursor at 0x1187a76c0>
```

- ❶ Создание таблицы из пяти столбцов, заполняемых вещественными числами (объектами `float`).

Теперь к данным, хранящимся в объекте `ndarray`, можно применить метод `executemany()`. Операции чтения/записи выполняются уже известным вам способом. Результаты запроса можно легко визуализировать (рис. 9.1).

```
In [95]: %%time
         qm('INSERT INTO numbers VALUES (?, ?, ?, ?, ?)', data) ❶
         con.commit()
         CPU times: user 7.3 s, sys: 195 ms, total: 7.49 s
         Wall time: 7.71 s

In [96]: ll $path*
         -rw-r--r--  1 yves  staff  52633600 Oct 19 12:11
         /Users/yves/Temp/data/numbers.db

In [97]: %%time
         temp = q('SELECT * FROM numbers').fetchall() ❷
         print(temp[:3])
```

```
[(0.4918, 1.3707, 0.137, 0.3981, -1.0059), (0.4516,
 1.4445, 0.0555, -0.0397, 0.44), (0.1629, -0.8473,
 -0.8223, -0.4621, -0.5137)]
CPU times: user 1.7 s, sys: 124 ms, total: 1.82 s
Wall time: 1.9 s
```

```
In [98]: %%time
query = 'SELECT * FROM numbers WHERE No1 > 0 AND No2 < 0'
res = np.array(q(query).fetchall()).round(3) ❸
CPU times: user 639 ms, sys: 64.7 ms, total: 704 ms
Wall time: 702 ms
```

```
In [99]: res = res[::100] ❹
plt.figure(figsize=(10, 6))
plt.plot(res[:, 0], res[:, 1], 'ro') ❺
```

- ❶ Вставка всего набора данных в таблицу одной командой.
- ❷ Извлечение всех строк таблицы одной командой.
- ❸ Выборка строк и преобразование их в объект `pandas`.
- ❹ Визуализация результатов выполнения запроса.

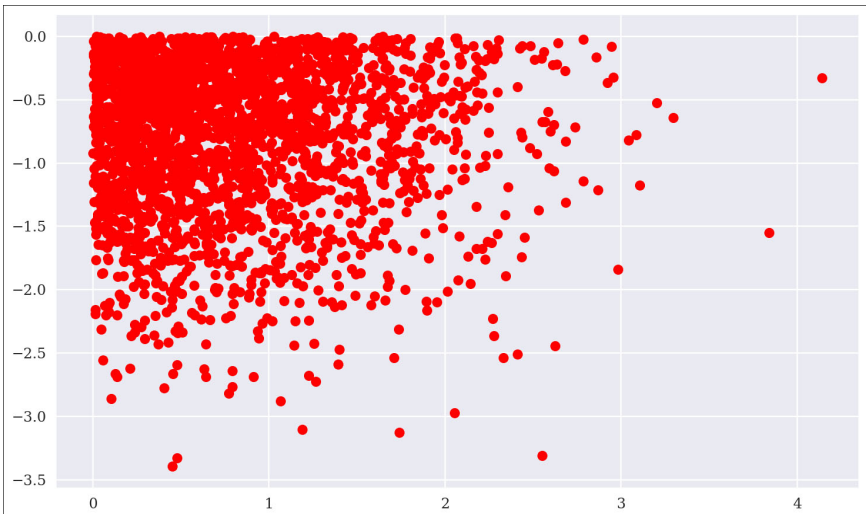


Рис. 9.1. Диаграмма рассеяния для результатов запроса (выборка)



## Импорт данных из реляционных баз данных

Все же более эффективный подход заключается в импорте таблиц или результатов запроса с помощью библиотеки `pandas`. Когда вся таблица находится в памяти, аналитические запросы выполняются намного быстрее, чем в случае обращения к файлам базы данных.

Чтение таблицы с помощью библиотеки `pandas` занимает примерно столько же времени, сколько занимает загрузка данных в объект `ndarray` библиотеки `NumPy`. В этом случае узким местом оказывается сама база данных.

```
In [100]: %time data = pd.read_sql('SELECT * FROM numbers', con) ❶
CPU times: user 2.17 s, sys: 180 ms, total: 2.35 s
Wall time: 2.32 s
```

```
In [101]: data.head()
```

```
Out[101]:
```

	No1	No2	No3	No4	No5
0	0.4918	1.3707	0.1370	0.3981	-1.0059
1	0.4516	1.4445	0.0555	-0.0397	0.4400
2	0.1629	-0.8473	-0.8223	-0.4621	-0.5137
3	1.3064	0.9125	0.5142	-0.7868	-0.3398
4	-0.1148	-1.5215	-0.7045	-1.0042	-0.0600

❶ Считывание всех строк таблицы в объект `DataFrame` с именем `data`.

Как только данные оказываются в памяти, их обработка резко ускоряется. Производительность часто возрастает на порядок или даже больше. Кроме того, с помощью библиотеки `pandas` можно составлять более сложные запросы, хотя они, конечно же, не могут служить полноценной заменой запросам SQL. На рис. 9.2 визуализируются результаты выполнения запроса с составным условием.

```
In [102]: %time data[(data['No1'] > 0) & (data['No2'] < 0)].head() ❶
CPU times: user 47.1 ms, sys: 12.3 ms, total: 59.4 ms
Wall time: 33.4 ms
```

```
Out[102]:
```

	No1	No2	No3	No4	No5
2	0.1629	-0.8473	-0.8223	-0.4621	-0.5137
5	0.1893	-0.0207	-0.2104	0.9419	0.2551
8	1.4784	-0.3333	-0.7050	0.3586	-0.3937
10	0.8092	-0.9899	1.0364	-1.0453	0.0579
11	0.9065	-0.7757	-0.9267	0.7797	0.0863

```
In [103]: %%time
          q = '(No1 < -0.5 | No1 > 0.5) & (No2 < -1 | No2 > 1)' ❷
          res = data[['No1', 'No2']].query(q) ❷
          CPU times: user 95.4 ms, sys: 22.4 ms, total: 118 ms
          Wall time: 56.4 ms
```

```
In [104]: plt.figure(figsize=(10, 6))
          plt.plot(res['No1'], res['No2'], 'ro');
```

- ❶ Два условия отбора записей.
- ❷ Четыре условия отбора записей.

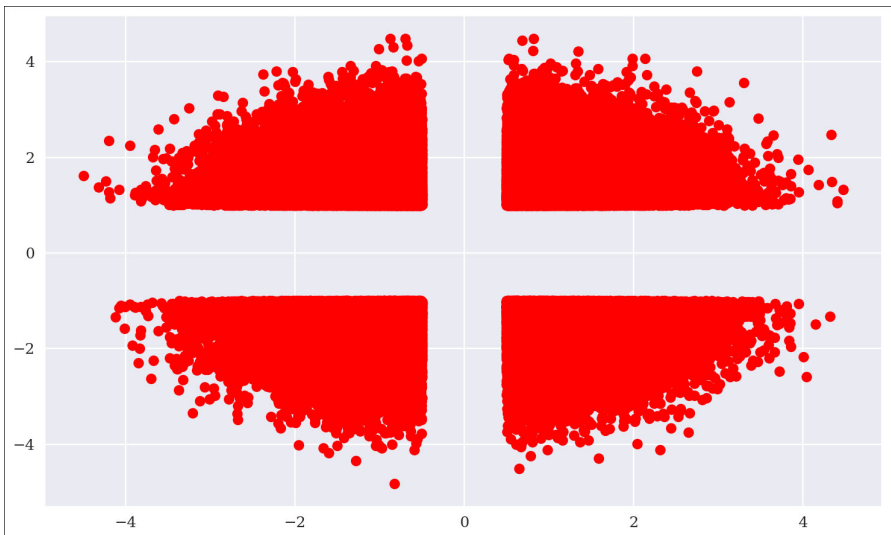


Рис. 9.2. Диаграмма рассеяния для результатов запроса (выборка)

Как и ожидалось, применение инструментов библиотеки `pandas`, обрабатывающих данные непосредственно в памяти, дает значительный прирост в производительности, при условии, что библиотека позволяет воспроизвести соответствующую инструкцию SQL.

Но это не единственное преимущество библиотеки, ведь она тесно интегрирована с другими пакетами, включая `PyTable`, с которым мы познакомимся далее. Пока достаточно знать, что применение пакетов в связке позволяет существенно ускорить выполнение операций ввода-вывода, как показано в следующем примере.

```
In [105]: h5s = pd.HDFStore(filename + '.h5s', 'w') ❶
```

```
In [106]: %time h5s['data'] = data ❷  
CPU times: user 46.7 ms, sys: 47.1 ms, total: 93.8 ms  
Wall time: 99.7 ms
```

```
In [107]: h5s ❸  
Out[107]: <class 'pandas.io.pytables.HDFStore'>  
File path: /Users/yves/Temp/data/numbers.h5s
```

```
In [108]: h5s.close() ❹
```

- ❶ Открытие базы данных HDF5 для записи. Библиотека `pandas` создает объект `HDFStore`, в котором будут храниться данные.
- ❷ Объект `DataFrame` сохраняется в базе данных в двоичном виде.
- ❸ Сведения об объекте `HDFStore`.
- ❹ Закрытие файла базы данных.

В данном случае объект `DataFrame`, хранящий все данные из исходной реляционной таблицы, записывается на диск намного быстрее, чем при использовании встроенных инструментов `SQLite3`. А чтение выполняется еще быстрее.

```
In [109]: %%time  
h5s = pd.HDFStore(filename + '.h5s', 'r') ❶  
data_ = h5s['data'] ❷  
h5s.close() ❸  
CPU times: user 11 ms, sys: 18.3 ms, total: 29.3 ms  
Wall time: 29.4 ms
```

```
In [110]: data_ is data ❹  
Out[110]: False
```

```
In [111]: (data_ == data).all() ❺  
Out[111]: No1    True  
          No2    True  
          No3    True  
          No4    True  
          No5    True  
          dtype: bool
```

```
In [112]: np.allclose(data_, data) ❺  
Out[112]: True
```

```
In [113]: ll $path* ❸
-rw-r--r-- 1 yves staff 52633600 Oct 19 12:11
/Users/yves/Temp/data/numbers.db
-rw-r--r-- 1 yves staff 48007240 Oct 19 12:11
/Users/yves/Temp/data/numbers.h5s
```

- ❶ Открытие файла базы данных HDF5 для чтения.
- ❷ Объект `DataFrame` загружается в память под именем `data_`.
- ❸ Закрытие файла базы данных.
- ❹ Два объекта `DataFrame` различаются...
- ❺ ...но при этом хранят одинаковые данные.
- ❻ Двоичный файл, как правило, меньше по размеру, чем файл базы данных.

## Работа с CSV-файлами

CSV — один из самых популярных форматов обмена финансовыми данными. Несмотря на отсутствие формального стандарта CSV-файлы могут обрабатываться на любых платформах и практически во всех приложениях, связанных с анализом и обработкой финансовых данных. Ранее мы уже узнали, как осуществлять чтение и запись таких файлов с помощью стандартных инструментов Python. В библиотеке `pandas` эти задачи выполняются немного проще, компактнее и быстрее (рис. 9.3).

```
In [114]: %time data.to_csv(filename + '.csv') ❶
CPU times: user 6.44 s, sys: 139 ms, total: 6.58 s
Wall time: 6.71 s
```

```
In [115]: ll $path
total 283672
-rw-r--r-- 1 yves staff 43834157 Oct 19 12:11 numbers.csv
-rw-r--r-- 1 yves staff 52633600 Oct 19 12:11 numbers.db
-rw-r--r-- 1 yves staff 48007240 Oct 19 12:11 numbers.h5s
```

```
In [116]: %time df = pd.read_csv(filename + '.csv') ❷
CPU times: user 1.12 s, sys: 111 ms, total: 1.23 s
Wall time: 1.23 s
```

```
In [117]: df[['No1', 'No2', 'No3', 'No4']].hist(bins=20,
figsize=(10, 6));
```

- ❶ Метод `.to_csv()` записывает данные объекта `DataFrame` на диск в формате CSV.
- ❷ Метод `pd.read_csv()` считывает созданный ранее CSV-файл в память, возвращая новый объект `DataFrame`.

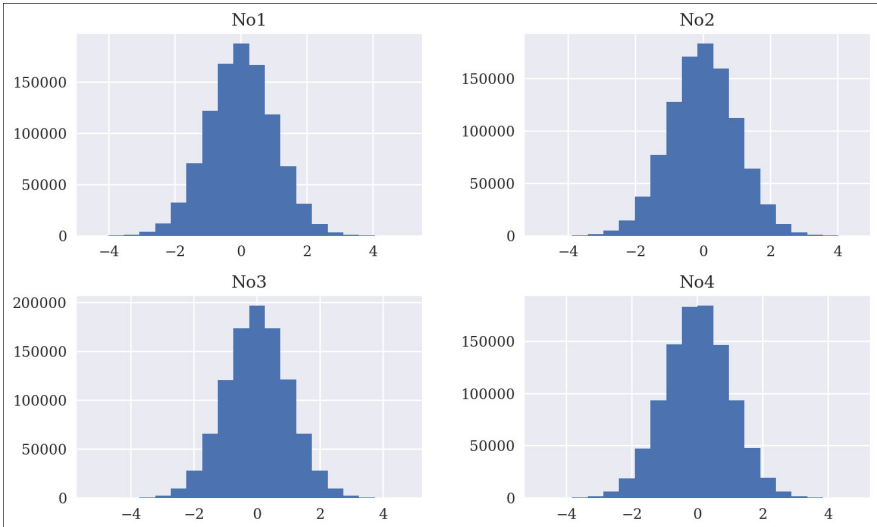


Рис. 9.3. Гистограммы отдельных столбцов

## Работа с файлами Excel

В следующем примере показано, как с помощью библиотеки `pandas` записать данные в формате Excel и загрузить данные из рабочих листов Excel. В данном случае набор данных содержит 100 000 строк (рис. 9.4).

```
In [118]: %time data[:100000].to_excel(filename + '.xlsx') ❶
          CPU times: user 25.9 s, sys: 520 ms, total: 26.4 s
          Wall time: 27.3 s
```

```
In [119]: %time df = pd.read_excel(filename + '.xlsx', 'Sheet1') ❷
          CPU times: user 5.78 s, sys: 70.1 ms, total: 5.85 s
          Wall time: 5.91 s
```

```
In [120]: df.cumsum().plot(figsize=(10, 6));
```

```
In [121]: ll $path*
          -rw-r--r--  1 yves  staff  43834157 Oct 19 12:11
```

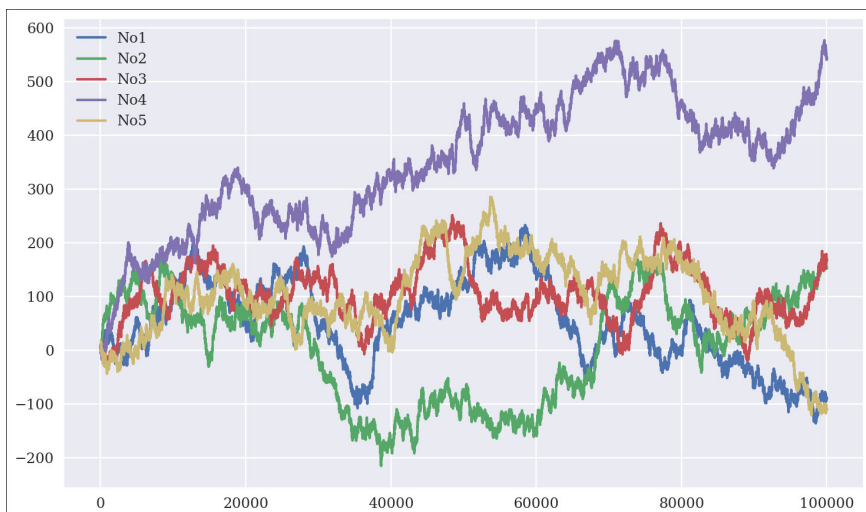
```

/Users/yves/Temp/data/numbers.csv
-rw-r--r-- 1 yves staff 52633600 Oct 19 12:11
/Users/yves/Temp/data/numbers.db
-rw-r--r-- 1 yves staff 48007240 Oct 19 12:11
/Users/yves/Temp/data/numbers.h5s
-rw-r--r-- 1 yves staff 4032725 Oct 19 12:12
/Users/yves/Temp/data/numbers.xlsx

```

In [122]: `rm -f $path*`

- ❶ Метод `.to_excel()` записывает данные объекта `DataFrame` на диск в формате XLSX.
- ❷ Метод `pd.read_excel()` считывает созданный ранее файл Excel в память, возвращая новый объект `DataFrame` (дополнительно указывается рабочий лист, из которого загружаются данные).



*Рис. 9.4. Графики всех столбцов данных*

Генерирование файла Excel, содержащего даже относительно небольшой объем данных, отнимает достаточно много времени. Это показывает, какие накладные расходы возникают при построении структуры рабочих листов.

## Ввод и вывод данных с помощью PyTables

PyTables — это пакет Python, предназначенный для работы с базами данных стандарта HDF5. Он специально разработан для оптимизации операций ввода-вывода с учетом имеющегося оборудования. Имя импортируемой библиотеки — `tables`. Как и библиотеку `pandas`, пакет PyTables нельзя считать полноценной заменой реляционным базам данных в том, что касается выполнения аналитических запросов в памяти. Тем не менее он содержит ряд полезных дополнений, упрощающих составление сложных запросов. К примеру, базы данных PyTables могут состоять из нескольких таблиц, а также поддерживают сжатие, индексацию и нестандартные запросы. К тому же они обеспечивают эффективное хранение массивов NumPy и поддерживают собственные структуры данных, близкие к массивам.

Для начала импортируем несколько библиотек.

```
In [123]: import tables as tb ❶  
         import datetime as dt
```

- ❶ Пакет называется PyTables, но в Python импортируется модуль `tables`.

### Работа с таблицами

Файловый формат базы данных PyTables напоминает формат SQLite<sup>6</sup>. В следующем примере открывается файл базы данных и создается таблица.

```
In [124]: filename = path + 'pytab.h5'
```

```
In [125]: h5 = tb.open_file(filename, 'w') ❶
```

```
In [126]: row_des = {  
            'Date': tb.StringCol(26, pos=1), ❷  
            'No1': tb.IntCol(pos=2), ❸  
            'No2': tb.IntCol(pos=3), ❸  
            'No3': tb.Float64Col(pos=4), ❹  
            'No4': tb.Float64Col(pos=5) ❹  
        }
```

```
In [127]: rows = 20000000
```

---

<sup>6</sup> Во многих СУБД реализуется архитектура “клиент/сервер”. Но для интерактивного анализа финансовых данных вполне достаточно файловых баз данных.

```

In [128]: filters = tb.Filters(complevel=0) ❸

In [129]: tab = h5.create_table('/', 'ints_floats', ❹
            row_des, ❺
            title='Integers and Floats', ❻
            expectedrows=rows, ❼
            filters=filters) ❽

In [130]: type(tab)
Out[130]: tables.table.Table

In [131]: tab
Out[131]: /ints_floats (Table(0,)) 'Integers and Floats'
        description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
        byteorder := 'little'
        chunkshape := (2621,)

```

- ❶ Открытие файла базы данных в двоичном формате HDF5.
- ❷ Столбец Date для значений даты и времени (данные хранятся в строковом виде).
- ❸ Два столбца для хранения целых чисел.
- ❹ Два столбца для хранения вещественных чисел.
- ❺ С помощью объектов Filters можно, в частности, задавать степень сжатия данных.
- ❻ Узел (путь расположения) и внутреннее имя таблицы
- ❼ Описание структуры записей.
- ❽ Имя (заголовок) таблицы.
- ❾ Ожидаемое количество строк (обеспечивает оптимизацию).
- ❽ Объект Filters, применяемый к таблице.

Для заполнения таблицы числовыми данными генерируются два объекта `ndarray`, содержащих случайные числа: один хранит целые числа, а другой — вещественные. Заполнение таблицы осуществляется в простом цикле.



```

In [132]: pointer = tab.row ❶

In [133]: ran_int = np.random.randint(0, 10000, size=(rows, 2)) ❷

In [134]: ran_flo = np.random.standard_normal((rows, 2)).round(4) ❸

In [135]: %%time
           for i in range(rows):
               pointer['Date'] = dt.datetime.now() ❹
               pointer['No1'] = ran_int[i, 0] ❹
               pointer['No2'] = ran_int[i, 1] ❹
               pointer['No3'] = ran_flo[i, 0] ❹
               pointer['No4'] = ran_flo[i, 1] ❹
               pointer.append() ❺
           tab.flush() ❻
CPU times: user 8.16 s, sys: 78.7 ms, total: 8.24 s
Wall time: 8.25 s

In [136]: tab ❼
Out[136]: /ints_floats (Table(2000000,)) 'Integers and Floats'
           description := {
               "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
               "No1": Int32Col(shape=(), dflt=0, pos=1),
               "No2": Int32Col(shape=(), dflt=0, pos=2),
               "No3": Float64Col(shape=(), dflt=0.0, pos=3),
               "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
           byteorder := 'little'
           chunkshape := (2621,)

In [137]: ll $path*
           -rw-r--r--  1 yves  staff  100156248 Oct 19 12:12
           /Users/yves/Temp/data/pytab.h5

```

- ❶ Создание объекта указателя.
- ❷ Генерирование объекта `ndarray`, содержащего случайные целые числа.
- ❸ Генерирование объекта `ndarray`, содержащего случайные вещественные числа.
- ❹ Построчная запись объекта `datetime`, двух объектов `int` и двух объектов `float`.
- ❺ Добавление новой строки в конец таблицы.

- ⑥ Очистка табличного буфера, что приводит к фиксации транзакции.
- ⑦ Просмотр описания табличного объекта.

В данном примере цикл Python работает очень медленно. Существует более удобный и производительный способ сделать то же самое с помощью структурированных массивов NumPy. Когда весь набор данных хранится в структурированном массиве, создать таблицу можно буквально одной строкой кода. Стоит отметить, что в этом случае уже нет необходимости создавать описание строк — в PyTables все необходимые типы данных определяются с помощью объекта `dtype` структурированного массива.

```
In [138]: dtype = np.dtype([('Date', 'S26'), ('No1', '<i4'),  
                           ('No2', '<i4'), ('No3', '<f8'),  
                           ('No4', '<f8')]) ①
```

```
In [139]: sarray = np.zeros(len(ran_int), dtype=dtype) ②
```

```
In [140]: sarray[:4] ③  
Out[140]: array([(b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.),  
                (b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.)],  
               dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'),  
                ('No3', '<f8'), ('No4', '<f8')])
```

```
In [141]: %%time  
          sarray['Date'] = dt.datetime.now() ④  
          sarray['No1'] = ran_int[:, 0] ④  
          sarray['No2'] = ran_int[:, 1] ④  
          sarray['No3'] = ran_flo[:, 0] ④  
          sarray['No4'] = ran_flo[:, 1] ④  
          CPU times: user 161 ms, sys: 42.7 ms, total: 204 ms  
          Wall time: 207 ms
```

```
In [142]: %%time  
          h5.create_table('/', 'ints_floats_from_array', sarray,  
                          title='Integers and Floats',  
                          expectedrows=rows, filters=filters) ⑤  
          CPU times: user 42.9 ms, sys: 51.4 ms, total: 94.3 ms  
          Wall time: 96.6 ms  
Out[142]: /ints_floats_from_array (Table(2000000,))  
          'Integers and Floats'  
          description := {  
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
```

```

"No1": Int32Col(shape=(), dflt=0, pos=1),
"No2": Int32Col(shape=(), dflt=0, pos=2),
"No3": Float64Col(shape=(), dflt=0.0, pos=3),
"No4": Float64Col(shape=(), dflt=0.0, pos=4)}
byteorder := 'little'
chunkshape := (2621,)

```

- ❶ Определение специального объекта `dtype`.
- ❷ Создание структурированного массива, заполненного нулями (и пустыми строками).
- ❸ Несколько записей объекта `ndarray`.
- ❹ Векторизованное заполнение столбцов объекта `ndarray`.
- ❺ Создание объекта `table` и заполнение его данными.

Как видите, мы сократили код, увеличили производительность на порядок и добились того же самого результата.

```
In [143]: type(h5)
```

```
Out[143]: tables.file.File
```

```
In [144]: h5 ❶
```

```

Out[144]: File(filename=/Users/yves/Temp/data/pytab.h5, title='',
mode='w', root_uep='/', filters=Filters(complevel=0,
shuffle=False, bitshuffle=False, fletcher32=False,
least_significant_digit=None))
/ (RootGroup) ''
/ints_floats (Table(2000000,)) 'Integers and Floats'
description := {
  "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
  "No1": Int32Col(shape=(), dflt=0, pos=1),
  "No2": Int32Col(shape=(), dflt=0, pos=2),
  "No3": Float64Col(shape=(), dflt=0.0, pos=3),
  "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
byteorder := 'little'
chunkshape := (2621,)
/ints_floats_from_array (Table(2000000,))
'Integers and Floats'
description := {
  "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
  "No1": Int32Col(shape=(), dflt=0, pos=1),
  "No2": Int32Col(shape=(), dflt=0, pos=2),
  "No3": Float64Col(shape=(), dflt=0.0, pos=3),

```

```
"No4": Float64Col(shape=(), dflt=0.0, pos=4)}  
byteorder := 'little'  
chunkshape := (2621,)
```

```
In [145]: h5.remove_node('/', 'ints_floats_from_array') ❷
```

- ❶ Описание файлового объекта, содержащего два табличных объекта.
- ❷ Удаление второй таблицы, содержащей повторяющиеся данные.

В большинстве случаев объект `Table` ведет себя точно так же, как и структурированные объекты `ndarray` библиотеки `NumPy` (рис. 9.5).

```
In [146]: tab[:3] ❶  
Out[146]: array([(b'2018-10-19 12:12:28.227771', 8576, 5991,  
                  -0.0528, 0.2468),  
                 (b'2018-10-19 12:12:28.227858', 2990, 9310,  
                  -0.0261, 0.3932),  
                 (b'2018-10-19 12:12:28.227868', 4400, 4823,  
                  0.9133, 0.2579)],  
                dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'),  
                        ('No3', '<f8'), ('No4', '<f8')])
```

```
In [147]: tab[:4]['No4'] ❷  
Out[147]: array([ 0.2468, 0.3932, 0.2579, -0.5582])
```

```
In [148]: %time np.sum(tab[:]['No3']) ❸  
CPU times: user 76.7 ms, sys: 74.8 ms, total: 151 ms  
Wall time: 152 ms  
Out[148]: 88.8542999999997
```

```
In [149]: %time np.sum(np.sqrt(tab[:]['No1'])) ❹  
CPU times: user 91 ms, sys: 57.9 ms, total: 149 ms  
Wall time: 164 ms  
Out[149]: 133349920.3689251
```

```
In [150]: %%time  
plt.figure(figsize=(10, 6))  
plt.hist(tab[:]['No3'], bins=30); ❺  
CPU times: user 328 ms, sys: 72.1 ms, total: 400 ms  
Wall time: 456 ms
```

- ❶ Выбор строк по индексу.
- ❷ Выбор значений столбца по индексу.

- ③ Применение универсальных функций NumPy.
- ④ Построение гистограммы столбца, хранящегося в объекте Table.

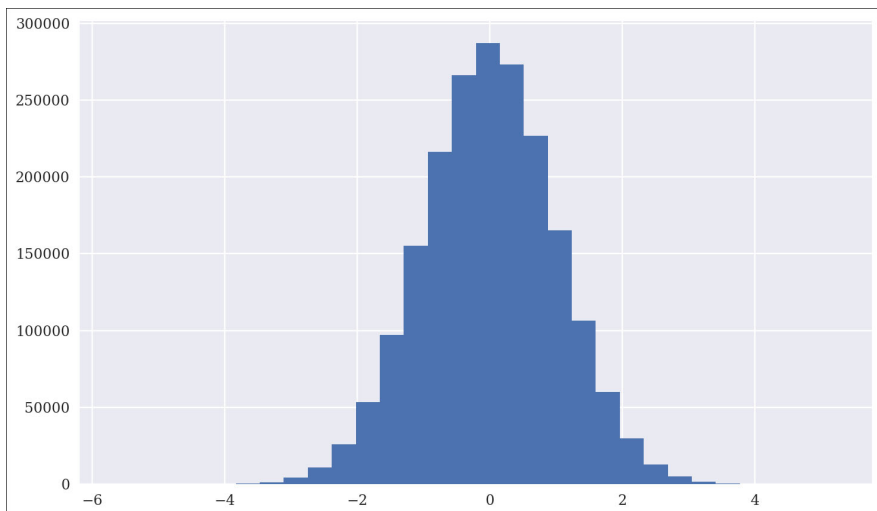


Рис. 9.5. Гистограмма столбца

В PyTable также имеются удобные средства составления запросов, напоминающих типичные инструкции SQL, как показано в следующем примере. Сравните результат выполнения одного из таких запросов (рис. 9.6) с результатом выполнения запроса, созданного ранее с помощью библиотеки `pandas` (см. рис. 9.2).

```
In [151]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) |
                                                    (No4 > 1))' ❶

In [152]: iterator = tab.where(query) ❷

In [153]: %time res = [(row['No3'], row['No4']) for row in iterator] ❸
CPU times: user 269 ms, sys: 64.4 ms, total: 333 ms
Wall time: 294 ms

In [154]: res = np.array(res) ❹
          res[:3]
Out[154]: array([[0.7694, 1.4866],
                  [0.9201, 1.3346],
                  [1.4701, 1.8776]])
```

```
In [155]: plt.figure(figsize=(10, 6))  
          plt.plot(res.T[0], res.T[1], 'ro');
```

- ❶ Запрос в виде строки с четырьмя условиями, соединяемыми с помощью логических операторов.
- ❷ Объект итератора для запроса.
- ❸ Извлечение результатов запроса с помощью спискового включения...
- ❹ ...и формирование объекта `ndarray`.

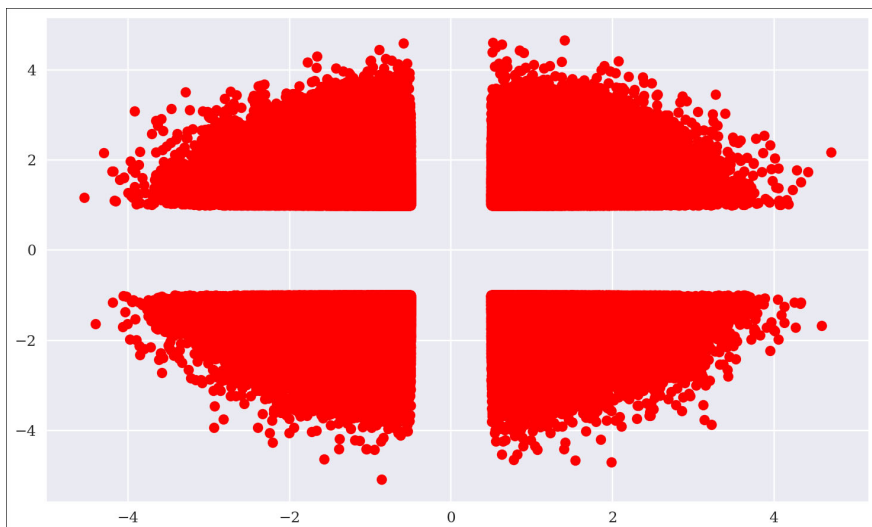


Рис. 9.6. Диаграмма рассеяния для столбцов



### Быстрые запросы

Библиотеки `pandas` и `PyTables` способны обрабатывать достаточно сложные SQL-подобные запросы. Обе библиотеки оптимизированы для выполнения таких операций. Несмотря на наличие определенных ограничений по сравнению с реляционными базами данных, для большинства аналитических и финансовых приложений эти ограничения совершенно несущественны.

Следующий пример позволяет убедиться в том, что работа с объектами `Table` библиотеки `PyTables` мало чем отличается от управления объектами `NumPy` и `pandas` с точки зрения как *синтаксиса*, так и *производительности*.

```
In [156]: %%time
values = tab[:, 'No3']
print('Max %18.3f' % values.max())
print('Mean %18.3f' % values.mean())
print('Min %18.3f' % values.min())
print('Std %18.3f' % values.std())
Max      5.224
Mean     0.000
Min     -5.649
Std      1.000
CPU times: user 163 ms, sys: 70.4 ms, total: 233 ms
Wall time: 234 ms
```

```
In [157]: %%time
res = [(row['No1'], row['No2']) for row in
        tab.where('((No1 > 9800) | (No1 < 200)) \
                    & ((No2 > 4500) & (No2 < 5500))')]
CPU times: user 165 ms, sys: 52.5 ms, total: 218 ms
Wall time: 155 ms
```

```
In [158]: for r in res[:4]:
            print(r)
(91, 4870)
(9803, 5026)
(9846, 4859)
(9823, 5069)
```

```
In [159]: %%time
res = [(row['No1'], row['No2']) for row in
        tab.where('(No1 == 1234) & (No2 > 9776)')]
CPU times: user 58.9 ms, sys: 40.5 ms, total: 99.4 ms
Wall time: 81 ms
```

```
In [160]: for r in res:
            print(r)
(1234, 9841)
(1234, 9821)
(1234, 9867)
(1234, 9987)
(1234, 9849)
(1234, 9800)
```

## Работа со сжатыми таблицами

Основное преимущество PyTables — поддержка сжатия, которое применяется не только для экономии места на диске, но и для ускорения операций ввода-вывода. За счет чего это достигается? Если скорость каналов ввода-вывода является узким местом и центральный процессор способен быстро обрабатывать сжатые потоки данных, то применение сжатия приводит к выигрышу в производительности. В следующих примерах данные хранятся на SSD-накопителе, поэтому заметных преимуществ по скорости не наблюдается, зато можно убедиться в отсутствии каких-либо *недостатков*, связанных с применением сжатия.

```
In [161]: filename = path + 'pytabc.h5'
```

```
In [162]: h5c = tb.open_file(filename, 'w')
```

```
In [163]: filters = tb.Filters(complevel=5, ❶  
                             complib='blosc') ❷
```

```
In [164]: tabc = h5c.create_table('/', 'ints_floats', sarray,  
                                title='Integers and Floats',  
                                expectedrows=rows, filters=filters)
```

```
In [165]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) |  
                                (No4 > 1))'
```

```
In [166]: iteratorc = tabc.where(query) ❸
```

```
In [167]: %time res = [(row['No3'], row['No4']) for row in  
                                iteratorc] ❹  
CPU times: user 300 ms, sys: 50.8 ms, total: 351 ms  
Wall time: 311 ms
```

```
In [168]: res = np.array(res)  
          res[:3]
```

```
Out[168]: array([[0.7694, 1.4866],  
                [0.9201, 1.3346],  
                [1.4701, 1.8776]])
```

- ❶ Параметр `complevel` задает степень сжатия в диапазоне от 0 (без сжатия) до 9 (максимальное сжатие).
- ❷ Применение оптимизированной системы сжатия Blosc (<http://blosc.org/>).



- ③ Создание объекта итератора для запроса.
- ④ Извлечение результатов запроса с помощью спискового включения.

Сжатый объект Table создается и обрабатывается чуть медленнее, чем несжатый. Теперь проверим, с какой скоростью выполняется загрузка данных в объект ndarray.

```
In [169]: %time arr_non = tab.read() ❶
          CPU times: user 63 ms, sys: 78.5 ms, total: 142 ms
          Wall time: 149 ms
```

```
In [170]: tab.size_on_disk
Out[170]: 100122200
```

```
In [171]: arr_non.nbytes
Out[171]: 100000000
```

```
In [172]: %time arr_com = tabc.read() ❷
          CPU times: user 106 ms, sys: 55.5 ms, total: 161 ms
          Wall time: 173 ms
```

```
In [173]: tabc.size_on_disk
Out[173]: 41306140
```

```
In [174]: arr_com.nbytes
Out[174]: 100000000
```

```
In [175]: ll $path* ❸
-rw-r--r-- 1 yves staff 200312336 Oct 19 12:12
/Users/yves/Temp/data/pytab.h5
-rw-r--r-- 1 yves staff 41341436 Oct 19 12:12
/Users/yves/Temp/data/pytabc.h5
```

```
In [176]: h5c.close() ❹
```

- ❶ Считывание несжатого объекта tab.
- ❷ Считывание сжатого объекта tabc.
- ❸ Сравнение размеров файлов: сжатая таблица занимает значительно меньше места.
- ❹ Заккрытие файла базы данных.

Эти примеры наглядно показывают, что особой разницы по скорости между сжатыми и несжатыми объектами Table нет. А вот размеры файлов на диске могут существенно отличаться (в зависимости от структуры данных), что дает целый ряд преимуществ:

- уменьшение стоимости хранения данных;
- уменьшение стоимости резервного копирования данных;
- уменьшение сетевого трафика;
- повышение скорости передачи данных по сети (ускорение обмена данными с сервером);
- устранение узких мест, связанных с пропускной способностью каналов ввода-вывода, за счет выполнения определенных операций центральным процессором.

## Работа с массивами

Как было показано в начале главы, библиотека NumPy обладает встроенными оптимизированными средствами чтения и записи данных в объекты `ndarray`. Пакет PyTables тоже реализует достаточно быстрые и эффективные операции сохранения и загрузки объектов `ndarray`, а поскольку он основан на иерархической структуре базы данных, это обеспечивает ряд дополнительных преимуществ.

```
In [177]: %%time
arr_int = h5.create_array('/', 'integers', ran_int) ❶
arr_flo = h5.create_array('/', 'floats', ran_flo) ❷
CPU times: user 4.26 ms, sys: 37.2 ms, total: 41.5 ms
Wall time: 46.2 ms
```

```
In [178]: h5 ❸
Out[178]: File(filename=/Users/yves/Temp/data/pytab.h5, title='',
mode='w', root_uep='/', filters=Filters(complevel=0,
shuffle=False, bitshuffle=False, fletcher32=False,
least_significant_digit=None))
/ (RootGroup) ''
/floats (Array(20000000, 2)) ''
atom := Float64Atom(shape=(), dflt=0.0)
maindim := 0
flavor := 'numpy'
byteorder := 'little'
chunkshape := None
```

```

/integers (Array(2000000, 2)) ''
  atom := Int64Atom(shape=(), dflt=0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
/ints_floats (Table(2000000,)) 'Integers and Floats'
  description := {
    "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
    "No1": Int32Col(shape=(), dflt=0, pos=1),
    "No2": Int32Col(shape=(), dflt=0, pos=2),
    "No3": Float64Col(shape=(), dflt=0.0, pos=3),
    "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
  byteorder := 'little'
  chunkshape := (2621,)

```

```

In [179]: ll $path*
-rw-r--r--  1 yves  staff  262344490 Oct 19 12:12
/Users/yves/Temp/data/pytab.h5
-rw-r--r--  1 yves  staff  41341436 Oct 19 12:12
/Users/yves/Temp/data/pytabc.h5

```

```
In [180]: h5.close()
```

```
In [181]: !rm -f $path*
```

- ❶ Сохранение объекта `ndarray` с именем `gan_int`.
- ❷ Сохранение объекта `ndarray` с именем `gan_flo`.
- ❸ Изменения отражаются в описании объекта.

Непосредственная запись таких объектов в базу данных HDF5 выполняется намного быстрее, чем построчная запись значений в объект `Table` или структурированный массив `ndarray`.



### Хранилища данных в формате HDF5

Формат иерархической базы данных HDF5 служит прекрасной альтернативой реляционным базам данных, особенно когда дело касается обработки структурированной числовой и финансовой информации. Как при автономном применении пакета `PyTables`, так и в связке с библиотекой `pandas` можно добиться практически максимальной для используемого оборудования производительности операций чтения и записи данных.

## Вычисления в условиях нехватки памяти

Пакет PyTables поддерживает операции OOM (Out-of-Memory — вне оперативной памяти), что позволяет реализовать обработку массивов, не помещающихся в памяти компьютера. Рассмотрим следующий пример, основанный на использовании класса EArray. Такого рода объекты могут расширяться в одном из измерений (построчно), тогда как количество столбцов (элементов каждой записи) должно оставаться неизменным.

```
In [182]: filename = path + 'earray.h5'
```

```
In [183]: h5 = tb.open_file(filename, 'w')
```

```
In [184]: n = 500 ❶
```

```
In [185]: ear = h5.create_earray('/', 'ear', ❷  
          atom=tb.Float64Atom(), ❸  
          shape=(0, n)) ❹
```

```
In [186]: type(ear)
```

```
Out[186]: tables.earray.EArray
```

```
In [187]: rand = np.random.standard_normal((n, n)) ❺  
          rand[:4, :4]
```

```
Out[187]: array([[ -1.25983231,  1.11420699,  0.1667485 ,  0.7345676 ],  
                 [-0.13785424,  1.22232417,  1.36303097,  0.13521042],  
                 [ 1.45487119, -1.47784078,  0.15027672,  0.86755989],  
                 [-0.63519366,  0.1516327 , -0.64939447, -0.45010975]])
```

```
In [188]: %%time  
          for _ in range(750):  
              ear.append(rand) ❻  
          ear.flush()  
          CPU times: user 814 ms, sys: 1.18 s, total: 1.99 s  
          Wall time: 2.53 s
```

```
In [189]: ear
```

```
Out[189]: /ear (EArray(375000, 500)) ''  
          atom := Float64Atom(shape=(), dtype=0.0)  
          maindim := 0  
          flavor := 'numpy'  
          byteorder := 'little'  
          chunkshape := (16, 500)
```

```
In [190]: ear.size_on_disk
Out[190]: 1500032000
```

- ❶ Фиксированное количество столбцов.
- ❷ Путь размещения и внутреннее имя объекта EArray.
- ❸ Атомарный объект dtype для отдельных значений.
- ❹ Структура объекта при инициализации (отсутствие строк, n столбцов).
- ❺ Объект ndarray, заполняемый случайными числами...
- ❻ ...которые добавляются многократно.

Для выполнения ООМ-вычислений, в которых не задействуются статистические функции, нужен еще один объект EArray такой же структуры. В пакет PyTables включен специальный модуль Expr, основанный на библиотеке numexpr (<https://numexpr.readthedocs.io/>), который предназначен для эффективной работы с числовыми выражениями. В следующем примере модуль Expr применяется для вычисления математического выражения, представленного уравнением 9.1, по отношению ко всему объекту EArray.

#### Уравнение 9.1. Пример математического выражения

$$y = 3 \sin(x) + \sqrt{|x|}.$$

Результаты сохраняются в объекте out типа EArray, а само выражение вычисляется поблочно.

```
In [191]: out = h5.create_earray('/', 'out',
                                atom=tb.Float64Atom(),
                                shape=(0, n))

In [192]: out.size_on_disk
Out[192]: 0

In [193]: expr = tb.Expr('3 * sin(ear) + sqrt(abs(ear))') ❶

In [194]: expr.set_output(out, append_mode=True) ❷

In [195]: %time expr.eval() ❸
CPU times: user 3.08 s, sys: 1.7 s, total: 4.78 s
Wall time: 4.03 s
Out[195]: /out (EArray(375000, 500)) ''
          atom := Float64Atom(shape=(), dtype=0.0)
```

```
maindim := 0
flavor := 'numpy'
byteorder := 'little'
chunkshape := (16, 500)
```

```
In [196]: out.size_on_disk
```

```
Out[196]: 1500032000
```

```
In [197]: out[0, :10]
```

```
Out[197]: array([-1.73369462,  3.74824436,  0.90627898,  2.86786818,
                  1.75424957, -0.91108973, -1.68313885,  1.29073295,
                  -1.68665599, -1.71345309])
```

```
In [198]: %time out_ = out.read() ④
```

```
CPU times: user 1.03 s, sys: 1.1 s, total: 2.13 s
```

```
Wall time: 2.22 s
```

```
In [199]: out_[0, :10]
```

```
Out[199]: array([-1.73369462,  3.74824436,  0.90627898,  2.86786818,
                  1.75424957, -0.91108973, -1.68313885,  1.29073295,
                  -1.68665599, -1.71345309])
```

- ① Преобразование строкового выражения в объект Expr.
- ② Вывод результатов в объект out типа EArray.
- ③ Вычисление выражения.
- ④ Загрузка в память всего объекта EArray.

Если учесть, что расчеты выполняются не в оперативной памяти, то скорость можно считать вполне приемлемой, тем более что пример запущен на обычном компьютере. Для сравнения оценим производительность, достигаемую с помощью модуля numexpr (рассматривается в главе 10), который выполняет все операции в памяти. Как видите, процесс ускоряется, но не на порядок.

```
In [200]: import numexpr as ne ①
```

```
In [201]: expr = '3 * sin(out_) + sqrt(abs(out_))' ②
```

```
In [202]: ne.set_num_threads(1) ③
```

```
Out[202]: 4
```

```
In [203]: %time ne.evaluate(expr)[0, :10] ④
```

```
CPU times: user 2.51 s, sys: 1.54 s, total: 4.05 s
```

```

Wall time: 4.94 s
Out[203]: array([-1.64358578, 0.22567882, 3.31363043, 2.50443549,
                4.27413965, -1.41600606, -1.68373023, 4.01921805,
                -1.68117412, -1.66053597])

In [204]: ne.set_num_threads(4) ⑤
Out[204]: 1

In [205]: %time ne.evaluate(expr)[0, :10] ⑥
CPU times: user 3.39 s, sys: 1.94 s, total: 5.32 s
Wall time: 2.96 s
Out[205]: array([-1.64358578, 0.22567882, 3.31363043, 2.50443549,
                4.27413965, -1.41600606, -1.68373023, 4.01921805,
                -1.68117412, -1.66053597])

In [206]: h5.close()

In [207]: !rm -f $path*

```

- ① Импорт модуля, предназначенного для вычисления математических выражений *в оперативной памяти*.
- ② Числовое выражение, представленное в строковом виде.
- ③ Задание одного потока вычислений.
- ④ Вычисление выражения в оперативной памяти в одном потоке.
- ⑤ Увеличение количества потоков до четырех.
- ⑥ Вычисление выражения в оперативной памяти в четырех потоках.

## Ввод и вывод данных с помощью TsTables

Пакет TsTables использует объекты PyTables для создания высокопроизводительного хранилища временных рядов. Основной сценарий его применения — однократная запись, многократное чтение. Такой подход типичен для финансовых приложений, где данные поступают в режиме реального времени или асинхронно и сохраняются на диске с целью последующего анализа. Это может происходить, например, в программе тестирования торговой стратегии на исторических данных, где из временного ряда приходится регулярно извлекать различные поднаборы. Важно, чтобы считывание таких данных происходило с минимальными задержками.

## Исходные данные

Как всегда, сначала необходимо сгенерировать тестовый набор данных, достаточно большой для демонстрации преимуществ пакета TsTables. В следующем примере создаются три достаточно длинных временных ряда, получаемых путем моделирования геометрического броуновского движения (об этом мы поговорим в главе 12).

```
In [208]: no = 5000000 ❶  
          co = 3 ❷  
          interval = 1. / (12 * 30 * 24 * 60) ❸  
          vol = 0.2 ❹  
  
In [209]: %%time  
          rn = np.random.standard_normal((no, co)) ❺  
          rn[0] = 0.0 ❻  
          paths = 100 * np.exp(np.cumsum(-0.5 * vol ** 2 * interval +  
                                         vol * np.sqrt(interval) * rn, axis=0)) ❼  
          paths[0] = 100 ❽  
          CPU times: user 869 ms, sys: 175 ms, total: 1.04 s  
          Wall time: 812 ms
```

- ❶ Количество временных периодов.
- ❷ Количество временных рядов.
- ❸ Временной интервал в виде доли года.
- ❹ Волатильность.
- ❺ Случайные числа с нормальным распределением вероятностей.
- ❻ Обнуление исходного набора случайных чисел.
- ❼ Моделирование по методу Эйлера.
- ❽ Начальные значения траекторий равны 100.

Поскольку пакет TsTables хорошо работает с объектами DataFrame библиотеки pandas, данные приводятся именно к такому виду (рис. 9.7).

```
In [210]: dr = pd.date_range('2019-1-1', periods=no, freq='1s')
```

```
In [211]: dr[-6:]  
Out[211]: DatetimeIndex(['2019-02-27 20:53:14', '2019-02-27 20:53:15',  
                        '2019-02-27 20:53:16', '2019-02-27 20:53:17',
```



```
'2019-02-27 20:53:18', '2019-02-27 20:53:19'],
dtype='datetime64[ns]', freq='S')
```

```
In [212]: df = pd.DataFrame(paths, index=dr, columns=['ts1', 'ts2', \
                                                    'ts3'])
```

```
In [213]: df.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5000000 entries, 2019-01-01 00:00:00 to
                2019-02-27 20:53:19

Freq: S
Data columns (total 3 columns):
ts1      float64
ts2      float64
ts3      float64
dtypes: float64(3)
memory usage: 152.6 MB
```

```
In [214]: df.head()
```

```
Out[214]:
```

	ts1	ts2	ts3
2019-01-01 00:00:00	100.000000	100.000000	100.000000
2019-01-01 00:00:01	100.018443	99.966644	99.998255
2019-01-01 00:00:02	100.069023	100.004420	99.986646
2019-01-01 00:00:03	100.086757	100.000246	99.992042
2019-01-01 00:00:04	100.105448	100.036033	99.950618

```
In [215]: df[::1000000].plot(figsize=(10, 6));
```

## Хранение данных

Пакет TsTables хранит финансовые временные ряды в виде специальной блочной структуры, позволяющей быстро извлекать произвольные поднаборы на основе заданного временного интервала. Для этого в пакет добавлена функция `create_ts()`. В следующем примере типы данных столбцов таблицы определяются с помощью класса `IsDescription` из пакета `PyTables`.

```
In [216]: import tstab as tstab
```

```
In [217]: class ts_desc(tb.IsDescription):
            timestamp = tb.Int64Col(pos=0) ❶
            ts1 = tb.Float64Col(pos=1) ❷
            ts2 = tb.Float64Col(pos=2) ❷
            ts3 = tb.Float64Col(pos=3) ❷
```



Рис. 9.7. Графики выбранных точек временного ряда

```
In [218]: h5 = tb.open_file(path + 'tstab.h5', 'w') ❸
```

```
In [219]: ts = h5.create_ts('/', 'ts', ts_desc) ❹
```

```
In [220]: %time ts.append(df) ❺
CPU times: user 1.36 s, sys: 497 ms, total: 1.86 s
Wall time: 1.29 s
```

```
In [221]: type(ts)
Out[221]: tstabables.tstable.TsTable
```

```
In [222]: ls -n $path
total 328472
-rw-r--r-- 1 501 20 157037368 Oct 19 12:13 tstab.h5
```

- ❶ Столбец временных меток.
- ❷ Столбцы числовых данных.
- ❸ Открытие файла базы данных HDF5 для записи.
- ❹ Создание объекта TsTable на основе объекта ts\_desc.
- ❺ Добавление данных из объекта DataFrame в объект TsTable.

## Извлечение данных

Запись данных с помощью пакета TsTables выполняется очень быстро, хоть и зависит от используемого оборудования. То же самое касается и считывания данных в память. Удобнее всего то, что при этом возвращается объект `DataFrame` (рис. 9.8).

```
In [223]: read_start_dt = dt.datetime(2019, 2, 1, 0, 0) ❶  
         read_end_dt = dt.datetime(2019, 2, 5, 23, 59) ❷  
  
In [224]: %time rows = ts.read_range(read_start_dt, read_end_dt) ❸  
CPU times: user 182 ms, sys: 73.5 ms, total: 255 ms  
Wall time: 163 ms
```

```
In [225]: rows.info()❹  
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 431941 entries, 2019-02-01 00:00:00 to  
                                         2019-02-05 23:59:00  
Data columns (total 3 columns):  
ts1      431941 non-null float64  
ts2      431941 non-null float64  
ts3      431941 non-null float64  
dtypes: float64(3)  
memory usage: 13.2 MB
```

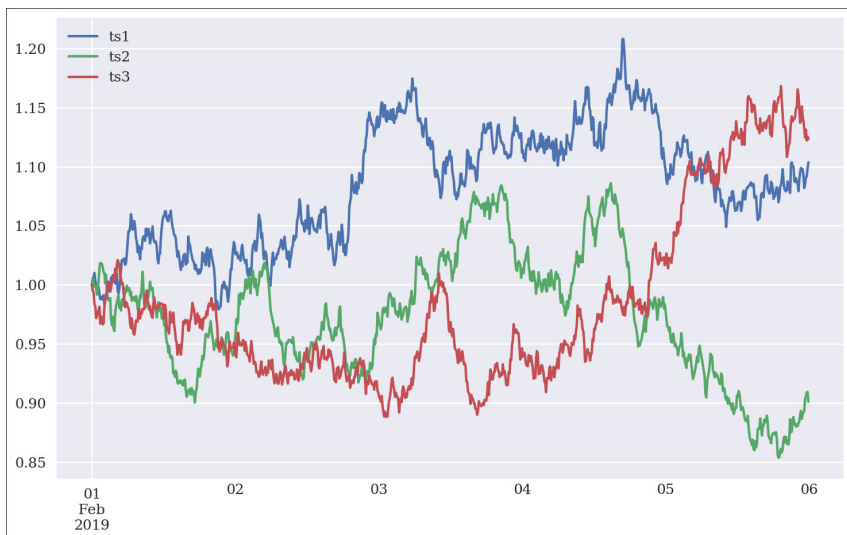
```
In [226]: rows.head()❹  
Out[226]:
```

		ts1	ts2	ts3
2019-02-01	00:00:00	52.063640	40.474580	217.324713
2019-02-01	00:00:01	52.087455	40.471911	217.250070
2019-02-01	00:00:02	52.084808	40.458013	217.228712
2019-02-01	00:00:03	52.073536	40.451408	217.302912
2019-02-01	00:00:04	52.056133	40.450951	217.207481

```
In [227]: h5.close()
```

```
In [228]: (rows[::500] / rows.iloc[0]).plot(figsize=(10, 6));
```

- ❶ Начало интервала.
- ❷ Конец интервала.
- ❸ Функция `ts.read_range()` возвращает объект `DataFrame`, соответствующий заданному интервалу.
- ❹ Объект `DataFrame` содержит несколько сотен тысяч строк данных.



*Рис. 9.8. Фрагмент финансового временного ряда (нормализованный) для заданного интервала*

Чтобы нагляднее продемонстрировать производительность операций чтения данных в пакете TsTables, рассмотрим следующий пример, в котором извлекаются 100 фрагментов данных, охватывающих трехдневный интервал с ежесекундной выборкой. На извлечение одного объекта DataFrame, содержащего 345 600 строк данных, уходит менее десятой доли секунды.

```
In [229]: import random
```

```
In [230]: h5 = tb.open_file(path + 'tstab.h5', 'r')
```

```
In [231]: ts = h5.root.ts._f_get_timeseries() ❶
```

```
In [232]: %%time
           for _ in range(100): ❷
               d = random.randint(1, 24) ❸
               read_start_dt = dt.datetime(2019, 2, d, 0, 0, 0)
               read_end_dt = dt.datetime(2019, 2, d + 3, 23, 59, 59)
               rows = ts.read_range(read_start_dt, read_end_dt)
           CPU times: user 7.17 s, sys: 1.65 s, total: 8.81 s
           Wall time: 4.78 s
```

```
In [233]: rows.info() ❹
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 345600 entries, 2019-02-04 00:00:00 to
                2019-02-07 23:59:59

Data columns (total 3 columns):
ts1      345600 non-null float64
ts2      345600 non-null float64
ts3      345600 non-null float64
dtypes: float64(3)
memory usage: 10.5 MB
```

```
In [234]: !rm $path/tstab.h5
```

- ❶ Подключение к объекту TsTables.
- ❷ Многократное извлечение данных.
- ❸ Случайный выбор начального дня.
- ❹ Информация о последнем извлеченном объекте DataFrame.

## Резюме

Реляционные базы данных обладают преимуществами с точки зрения структурирования хранимой в них информации и установки отношений между отдельными объектами/таблицами. В ряде случаев это оправдывает снижение производительности по сравнению с применением массивов `ndarray` библиотеки `NumPy` или объектов `DataFrame` библиотеки `pandas`.

Но во многих финансовых и научных приложениях вполне приемлемо работать с данными, хранящимися в виде массивов. В этом случае можно добиться огромного прироста производительности за счет использования встроенных средств ввода-вывода библиотеки `NumPy`, комбинации возможностей `NumPy` и `PyTables` или применения инструментов библиотеки `pandas`, ориентированных на управление базами данных формата `HDF5`. Пакет `TsTables` особенно полезен при работе с большими финансовыми временными рядами в рамках подхода “однократная запись, многократное чтение”.

В последние годы все большую популярность приобретают облачные решения, в которых облако образуется кластером готовых вычислительных систем. В такого рода архитектурах следует заранее продумывать, какие аппаратные платформы лучше всего подойдут для решения задач финансового анализа. Определенный свет на проблему проливает исследование, проведенное компанией `Microsoft`.

Мы утверждаем, что одиночный “вертикально масштабируемый” сервер позволяет решать все эти задачи, как минимум не уступая облачному кластеру с точки зрения производительности, стоимости, энергопотребления и компактности.

Аппусвами (2013)

Поэтому IT-компании, исследовательские институты и другие организации, занимающиеся анализом данных, должны сначала определиться с кругом решаемых задач и уже на основании этого выбрать соответствующую аппаратную/программную архитектуру, отталкиваясь от следующих двух парадигм.

#### *Горизонтальное масштабирование*

Предполагает использование кластера со множеством вычислительных узлов, каждый из которых оснащен стандартным процессором и не-большим объемом оперативной памяти.

#### *Вертикальное масштабирование*

Предполагает использование одного или нескольких высокопроизводительных серверов, снабженных многоядерными процессорами, специализированными графическими или даже тензорными процессорами (применяются в задачах машинного и глубокого обучения) и большим объемом оперативной памяти.

Выбор архитектуры может оказывать существенное влияние на производительность, о чем мы поговорим в следующей главе.

## **Дополнительные ресурсы**

В Интернете есть множество полезных ресурсов, посвященных рассмотренным в этой главе библиотекам и пакетам Python.

- Принципы сериализации объектов Python с помощью модуля `pickle` описаны в официальной документации (<https://docs.python.org/3/library/pickle.html>).
- Обзор средств ввода-вывода библиотеки NumPy дан на официальном сайте (<https://docs.scipy.org/doc/numpy/reference/routines.io.html>).
- Средства ввода-вывода, имеющиеся в библиотеке `pandas`, детально рассмотрены в онлайн-документации ([http://pandas.pydata.org/pandas-docs/stable/user\\_guide/io.html](http://pandas.pydata.org/pandas-docs/stable/user_guide/io.html)).

- Детальная документация к пакету PyTables доступна на официальном сайте ([www.pytables.org](http://www.pytables.org)).
- Информацию о пакете TsTables можно найти на GitHub (<http://github.com/afiedler/tstables/>).

Форк-версия пакета TsTables доступна по адресу <http://github.com/yhilpisch/tstables>. Для установки этой версии выполните следующую команду:

```
install git+git://github.com/yhilpisch/tstables
```

Данный пакет поддерживается для совместимости с последними версиями `pandas` и других библиотек Python.

Статья, процитированная в начале и конце главы, послужит хорошей отправной точкой при изучении вопроса о выборе аппаратной архитектуры для решения задач финансового анализа.

- Appuswamy, Raja, et al. *Nobody Ever Got Fired for Buying a Cluster* (2013; <http://bit.ly/2RZ0pR8>).

---

# Производительность Python

Не снижайте требования, чтобы получить нужную производительность.  
Повышайте производительность, чтобы удовлетворить требованиям.

*Ральф Марстон*

Существует устойчивое предубеждение, что Python сам по себе — относительно медленный язык программирования, который мало подходит для решения трудоемких финансовых задач. Помимо того факта, что Python — интерпретируемый язык, слабая производительность обычно объясняется низкой скоростью выполнения циклов, а поскольку циклы широко применяются в финансовых алгоритмах, значит, в Python эти алгоритмы реализуются медленно. Считается, что в компилируемых языках (такие, как C и C++) циклы выполняются значительно быстрее, а раз так, то они лучше подходят для реализации финансовых алгоритмов.

Безусловно, всегда можно написать правильный код Python, который будет выполняться медленно — даже слишком медленно для многих прикладных задач. В этой главе мы поговорим о том, как повысить производительность кода Python и реализовать более эффективные финансовые алгоритмы. Вы узнаете, что при использовании правильных структур данных, пакетов и методик программирования можно добиться от Python не меньшей производительности, чем у компилируемых языков. Это достигается в том числе за счет (частичной) компиляции кода.

В главе рассматривается несколько подходов к повышению производительности кода.

## *Векторизация вычислений*

Векторизация уже применялась нами в предыдущих главах.

## *Динамическая компиляция*

Для динамической компиляции чистого кода Python по технологии LLVM (<https://llvm.org/>) применяется пакет Numba.



### *Статическая компиляция*

Cython — это не только пакет Python, но и гибридный язык программирования, объединяющий Python и C. В нем, в частности, разрешены статические объявления типов данных и возможна статическая компиляция подобного кода.

### *Параллельная обработка данных*

Модуль `multiprocessing` позволяет легко распараллеливать выполнение кода.

В главе рассматриваются следующие темы.

### *Циклы*

Из этого раздела вы узнаете о том, как увеличить скорость выполнения циклов.

### *Алгоритмы*

В этом разделе рассматриваются стандартные математические алгоритмы, наиболее часто применяемые для оценки производительности, в частности алгоритм генерирования чисел Фибоначчи.

### *Биномиальные деревья*

Биномиальная модель оценки опционов широко используется в финансовых приложениях, поэтому с нее лучше всего начинать изучение более сложных финансовых алгоритмов.

### *Метод Монте-Карло*

Моделирование по методу Монте-Карло тоже широко задействуется в финансовой практике для управления рисками и ценообразованием. Такие задачи требуют высокой производительности вычислений и традиционно считались сферой применения таких языков, как C и C++.

### *Рекурсивный алгоритм библиотеки `pandas`*

В этом разделе обсуждаются способы повышения производительности рекурсивных алгоритмов, отвечающих за обработку финансовых временных рядов. В частности, мы рассмотрим несколько реализаций алгоритма вычисления экспоненциально взвешенного скользящего среднего (EWMA).

## **Циклы**

Этот раздел посвящен проблеме циклов Python. Задача очень простая: необходимо написать функцию, генерирующую большой набор случайных чисел и возвращающую его среднее значение. В первую очередь нас интересует

время выполнения функции, определяемое с помощью магических команд `%time` и `%timeit`.

## Python

Начнем с медленного решения. На чистом Python функция `average_py()` может выглядеть следующим образом.

```
In [1]: import random
```

```
In [2]: def average_py(n):  
        s = 0 ❶  
        for i in range(n):  
            s += random.random() ❷  
        return s / n ❸
```

```
In [3]: n = 10000000 ❹
```

```
In [4]: %time average_py(n) ❺  
CPU times: user 1.82 s, sys: 10.4 ms, total: 1.83 s  
Wall time: 1.93 s  
Out[4]: 0.5000590124747943
```

```
In [5]: %timeit average_py(n) ❻  
1.31 s ± 159 ms per loop (mean ± std. dev. of 7 runs,  
1 loop each)
```

```
In [6]: %time sum([random.random() for _ in range(n)]) / n ❼  
CPU times: user 1.55 s, sys: 188 ms, total: 1.74 s  
Wall time: 1.74 s  
Out[6]: 0.49987031710661173
```

- ❶ Инициализация переменной `s`.
- ❷ Генерирование случайных чисел из интервала (0, 1) с нормальным распределением и добавление их в переменную `s`.
- ❸ Функция возвращает среднее значение по набору.
- ❹ Определение количества итераций цикла.
- ❺ Однократный вызов функции и замер времени ее выполнения.
- ❻ Многократный вызов функции для более точного замера времени ее выполнения.
- ❼ Использование спискового включения вместо функции.

Полученный результат послужит эталоном, с которым мы будем сравнивать остальные решения.

## NumPy

Главное преимущество пакета NumPy заключается в векторизации вычислений. На уровне кода Python циклы здесь отсутствуют. Они реализуются на низком уровне внутри оптимизированных скомпилированных подпрограмм, предоставляемых библиотекой NumPy<sup>1</sup>. Рассмотрим следующую реализацию функции `average_py()`.

```
In [7]: import numpy as np
```

```
In [8]: def average_np(n):  
        s = np.random.random(n) ❶  
        return s.mean() ❷
```

```
In [9]: %time average_np(n)  
CPU times: user 180 ms, sys: 43.2 ms, total: 223 ms  
Wall time: 224 ms  
Out[9]: 0.49988861556468317
```

```
In [10]: %timeit average_np(n)  
128 ms ± 2.01 ms per loop (mean ± std. dev. of 7 runs,  
10 loops each)
```

```
In [11]: s = np.random.random(n)  
        s.nbytes ❸  
Out[11]: 800000000
```

- ❶ Получение случайных чисел за один проход (без использования циклов Python).
- ❷ Возврат среднего значения.
- ❸ Количество байтов, выделенных для хранения объекта `ndarray`.

Как видите, прирост производительности огромен: в 10 раз, т.е. на порядок! Но это достигается за счет существенного увеличения объема потребляемой памяти, что связано с необходимостью заранее выделить память под хранение данных, обрабатываемых компилятором. Как следствие, такой под-

---

<sup>1</sup> В NumPy могут также использоваться специализированные математические библиотеки, такие как Intel Math Kernel Library (<https://software.intel.com/en-us/mkl>).

ход неприменим для работы с потоковыми данными. В отдельных алгоритмах и задачах объем требуемой памяти может даже превысить имеющиеся ресурсы системы.



### Векторизация и оперативная память

Существует большой соблазн использовать векторизованный код NumPy везде, где только можно, ведь это позволяет сократить объем кода и повысить производительность. Но следует всегда учитывать и обратную сторону медали: объем используемой оперативной памяти резко увеличится.

## Numba

Numba (<https://numba.pydata.org/>) — это пакет, обеспечивающий *динамическую компиляцию* кода Python по технологии LLVM. В простых задачах, как в нашем случае, все реализуется на удивление легко. Динамически компилируемую функцию `average_nb()` можно вызвать напрямую из кода Python.

```
In [12]: import numba
```

```
In [13]: average_nb = numba.jit(average_py) ❶
```

```
In [14]: %time average_nb(n) ❷
```

```
CPU times: user 204 ms, sys: 34.3 ms, total: 239 ms  
Wall time: 278 ms
```

```
Out[14]: 0.4998865391283664
```

```
In [15]: %time average_nb(n) ❸
```

```
CPU times: user 80.9 ms, sys: 457 µs, total: 81.3 ms  
Wall time: 81.7 ms
```

```
Out[15]: 0.5001357454250273
```

```
In [16]: %timeit average_nb(n) ❸
```

```
75.5 ms ± 1.95 ms per loop (mean ± std. dev. of 7 runs,  
10 loops each)
```

- ❶ Создание функции Numba.
- ❷ Компиляция происходит в процессе выполнения кода, что связано с определенными накладными расходами.
- ❸ При повторном вызове функция выполняется значительно быстрее.

Сочетание чистого кода Python и пакета Numba обеспечивает более высокую производительность по сравнению с пакетом NumPy, и при этом память используется так же эффективно, как и в исходной версии с циклами. Самое удобное то, что никакой новый код писать не пришлось.



### Бесплатный сыр в мышеловке

Многим пакет Numba может показаться палочкой-выручалочкой, если учесть простоту его использования и выигрыш в производительности, достигаемый за счет компиляции кода. Но не спешите радоваться: во многих задачах этот пакет неприменим либо не позволяет получить никакого прироста производительности.

## Cython

Пакет Cython (<https://cython.org/>) обеспечивает *статическую компиляцию* кода Python. Здесь все не так просто, как в случае Numba, поскольку для получения существенного прироста производительности необходимо переписывать код. Для начала рассмотрим функцию `average_cy1()`, в которой применяется статическая типизация.

```
In [17]: %load_ext Cython
```

```
In [18]: %%cython -a
import random ❶
def average_cy1(int n): ❷
    cdef int i ❷
    cdef float s = 0 ❷
    for i in range(n):
        s += random.random()
    return s / n
```

```
Out[18]: <IPython.core.display.HTML object>
```

```
In [19]: %time average_cy1(n)
CPU times: user 695 ms, sys: 4.31 ms, total: 699 ms
Wall time: 711 ms
Out[19]: 0.49997106194496155
```

```
In [20]: %timeit average_cy1(n)
752 ms ± 91.1 ms per loop (mean ± std. dev. of 7 runs,
1 loop each)
```

- ❶ Импорт модуля `random` из пакета Cython.
- ❷ Статическое объявление типов переменных `n`, `i` и `s`.

Производительность повысилась, но не столь существенно, как в случае пакета NumPy. Чтобы добиться от Cython более ощутимых результатов (превосходящих даже версию Numba), нужно дополнительно оптимизировать код.

```
In [21]: %%cython
         from libc.stdlib cimport rand ❶
         cdef extern from 'limits.h': ❷
             int INT_MAX ❷
         cdef int i
         cdef float rn
         for i in range(5):
             rn = rand() / INT_MAX ❸
             print(rn)
0.6792964339256287
0.934692919254303
0.3835020661354065
0.5194163918495178
0.8309653401374817
```

```
In [22]: %%cython -a
         from libc.stdlib cimport rand ❶
         cdef extern from 'limits.h': ❷
             int INT_MAX ❷
         def average_cy2(int n):
             cdef int i
             cdef float s = 0
             for i in range(n):
                 s += rand() / INT_MAX ❸
             return s / n
```

Out[22]: <IPython.core.display.HTML object>

```
In [23]: %time average_cy2(n)
CPU times: user 78.5 ms, sys: 422 µs, total: 79 ms
Wall time: 79.1 ms
```

Out[23]: 0.500017523765564

```
In [24]: %timeit average_cy2(n)
65.4 ms ± 706 µs per loop (mean ± std. dev. of 7 runs,
10 loops each)
```

- ❶ Импорт генератора случайных чисел языка C.
- ❷ Импорт константы для масштабирования случайных чисел.
- ❸ Получение набора случайных чисел из интервала (0, 1) с нормальным распределением и применение к ним масштабирования.

Как видите, улучшенная версия исходной функции — `average_cy2()` — работает быстрее, чем версия, реализованная с помощью NumPy. Правда, это потребовало написания дополнительного кода. Самое главное, что пакет Cython управляет памятью намного эффективнее, чем пакет NumPy, сохраняя эффективность оригинальной реализации с циклами.



### Cython = Python + C

Cython позволяет разработчикам самостоятельно определять степень оптимизации кода. Оптимальным решением будет начать с чистого кода Python, добавляя в него элементы языка C только по мере необходимости. Процесс компиляции тоже можно параметризовать, чтобы дополнительно оптимизировать скомпилированную версию кода.

## Алгоритмы

В этом разделе мы применим изученные выше методики к известным математическим задачам и алгоритмам, которые часто используются для оценки производительности.

### Простые числа

Простые числа играют важную роль не только в фундаментальной математике, но и во многих прикладных областях, например в шифровании. *Простым* считается целое положительное число, большее единицы, которое делится без остатка на само себя и на 1. Его нельзя разложить на множители. Поиск больших простых чисел — довольно трудоемкая задача ввиду их редкости, зато относительно несложно доказать, что число не является простым. Нужно лишь найти множитель, на который число делится без остатка.

### Python

Существует множество алгоритмов для проверки простоты числа. Ниже приведено решение, которое не оптимально с алгоритмической точки зрения, зато достаточно эффективно. Тем не менее проверка большого числа `p2` занимает много времени.

```
In [25]: def is_prime(I):  
         if I % 2 == 0: return False ❶  
         for i in range(3, int(I ** 0.5) + 1, 2): ❷
```

```

        if I % i == 0: return False ❸
    return True ❹

In [26]: n = int(1e8 + 3) ❺
        n
Out[26]: 100000003

In [27]: %time is_prime(n)
CPU times: user 35 µs, sys: 0 ns, total: 35 µs
Wall time: 39.1 µs
Out[27]: False

In [28]: p1 = int(1e8 + 7) ❺
        p1
Out[28]: 100000007

In [29]: %time is_prime(p1)
CPU times: user 776 µs, sys: 1 µs, total: 777 µs
Wall time: 787 µs
Out[29]: True

In [30]: p2 = 100109100129162907G

In [31]: p2.bit_length() ❻
Out[31]: 57

In [32]: %time is_prime(p2)
CPU times: user 22.6 s, sys: 44.7 ms, total: 22.6 s
Wall time: 22.7 s
Out[32]: True

```

- ❶ Если число четное, сразу же возвращается False.
- ❷ Цикл начинается с 3 и продолжается до квадратного корня из I плюс 1 с шагом 2.
- ❸ При нахождении множителя возвращается False.
- ❹ Если множитель не найден, возвращается True.
- ❺ Проверяем относительно небольшие простые и составные числа.
- ❻ Большое простое число, проверка которого занимает много времени.



## Numba

Циклический алгоритм, реализованный в функции `is_prime()`, хорошо оптимизируется за счет динамической компиляции с помощью пакета Numba. Накладные расходы оказываются минимальными, а выигрыш в производительности — существенным.

```
In [33]: is_prime_nb = numba.jit(is_prime)
```

```
In [34]: %time is_prime_nb(n) ❶
```

```
CPU times: user 87.5 ms, sys: 7.91 ms, total: 95.4 ms
```

```
Wall time: 93.7 ms
```

```
Out[34]: False
```

```
In [35]: %time is_prime_nb(n) ❷
```

```
CPU times: user 9 µs, sys: 1e+03 ns, total: 10 µs
```

```
Wall time: 13.6 µs
```

```
Out[35]: False
```

```
In [36]: %time is_prime_nb(p1)
```

```
CPU times: user 26 µs, sys: 0 ns, total: 26 µs
```

```
Wall time: 31 µs
```

```
Out[36]: True
```

```
In [37]: %time is_prime_nb(p2) ❸
```

```
CPU times: user 1.72 s, sys: 9.7 ms, total: 1.73 s
```

```
Wall time: 1.74 s
```

```
Out[37]: True
```

- ❶ При первом вызове функции `is_prime_nb()` возникают накладные расходы, связанные с компиляцией.
- ❷ При втором вызове функции `is_prime_nb()` ускорение становится очевидным.
- ❸ В случае больших простых чисел скорость увеличивается на порядок.

## Cython

Применить пакет Cython в нашем случае несложно. Даже без объявления типов наблюдается существенный прирост производительности.

```
In [38]: %%cython
```

```
def is_prime_cy1(I):
```

```
    if I % 2 == 0: return False
```

```

    for i in range(3, int(I ** 0.5) + 1, 2):
        if I % i == 0: return False
    return True

```

```

In [39]: %timeit is_prime(p1)
394 µs ± 14.7 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)

```

```

In [40]: %timeit is_prime_cy1(p1)
243 µs ± 6.58 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)

```

Но настоящее ускорение достигается только при статической типизации. В этом случае версия Cython может даже работать чуть быстрее, чем версия Numba.

```

In [41]: %%cython
def is_prime_cy2(long I): ❶
    cdef long i ❶
    if I % 2 == 0: return False
    for i in range(3, int(I ** 0.5) + 1, 2):
        if I % i == 0: return False
    return True

```

```

In [42]: %timeit is_prime_cy2(p1)
87.6 µs ± 27.7 µs per loop (mean ± std. dev. of 7 runs,
10000 loops each)

```

```

In [43]: %time is_prime_nb(p2)
CPU times: user 1.68 s, sys: 9.73 ms, total: 1.69 s
Wall time: 1.7 s
Out[43]: True

```

```

In [44]: %time is_prime_cy2(p2)
CPU times: user 1.66 s, sys: 9.47 ms, total: 1.67 s
Wall time: 1.68 s
Out[44]: True

```

❶ Статическое объявление типов для переменных I и i.

## Параллельные вычисления

Пока что все наши усилия по оптимизации кода были сосредоточены на последовательной обработке данных. В тестах простоты имеет смысл одно-

временно обрабатывать сразу несколько чисел. В этом нам поможет модуль `multiprocessing` (<https://docs.python.org/3/library/multiprocessing.html>), позволяющий запускать несколько параллельных процессов Python. В нашем случае применить его несложно. Сначала необходимо создать объект `mp.Pool` для управления пулом процессов. Затем к пулу подключается выполняемая функция, которой передается список проверяемых чисел.

```
In [45]: import multiprocessing as mp
```

```
In [46]: pool = mp.Pool(processes=4) ❶
```

```
In [47]: %time pool.map(is_prime, 10 * [p1]) ❷  
CPU times: user 1.52 ms, sys: 2.09 ms, total: 3.61 ms  
Wall time: 9.73 ms
```

```
Out[47]: [True, True, True, True, True, True, True, True, True]
```

```
In [48]: %time pool.map(is_prime_nb, 10 * [p2]) ❷  
CPU times: user 13.9 ms, sys: 4.8 ms, total: 18.7 ms  
Wall time: 10.4 s
```

```
Out[48]: [True, True, True, True, True, True, True, True, True]
```

```
In [49]: %time pool.map(is_prime_cy2, 10 * [p2]) ❷  
CPU times: user 9.8 ms, sys: 3.22 ms, total: 13 ms  
Wall time: 9.51 s
```

```
Out[49]: [True, True, True, True, True, True, True, True, True]
```

- ❶ Инициализация объекта `mp.Pool` для управления пулом процессов.
- ❷ Подключение целевой функции и передача ей списка простых чисел.

Скорость вычислений возрастает очень сильно. Проверка большого простого числа `p2` с помощью функции `is_prime()`, написанной на чистом Python, заняла более 20 секунд, тогда как проверка десяти таких чисел с помощью функций `is_prime_nb()` и `is_prime_cy2()` занимает около 10 секунд за счет параллельного выполнения четырех процессов.



### Параллельные вычисления

К параллельной обработке следует прибегать при решении нескольких однотипных задач с разными входными значениями. Ускорение вычислений может оказаться огромным при наличии мощного многоядерного процессора и достаточного объема оперативной памяти. Модуль `multiprocessing` — это эффективный инструмент, включенный в состав стандартной библиотеки Python.

## Числа Фибоначчи

Числа Фибоначчи можно генерировать с помощью простого алгоритма. Первые два числа последовательности — 0 и 1, а каждое следующее число равно сумме двух предыдущих: 0, 1, 1, 2, 3, 5, 8, 13, 21... В этом разделе мы рассмотрим две реализации алгоритма: рекурсивную и итеративную.

### Рекурсивный алгоритм

Как и циклы, рекурсивные функции, написанные на чистом Python, работают достаточно медленно. Такие функции вызывают сами себя множество раз для получения конечного результата. Ниже приведена реализация рекурсивной функции `fib_rec_py1()`. В данном случае использование пакета Numba мало что дает, в отличие от пакета Cython, показывающего существенное ускорение за счет одной только статической типизации.

```
In [50]: def fib_rec_py1(n):  
        if n < 2:  
            return n  
        else:  
            return fib_rec_py1(n - 1) + fib_rec_py1(n - 2)
```

```
In [51]: %time fib_rec_py1(35)  
CPU times: user 6.55 s, sys: 29 ms, total: 6.58 s  
Wall time: 6.6 s  
Out[51]: 9227465
```

```
In [52]: fib_rec_nb = numba.jit(fib_rec_py1)
```

```
In [53]: %time fib_rec_nb(35)  
CPU times: user 3.87 s, sys: 24.2 ms, total: 3.9 s  
Wall time: 3.91 s  
Out[53]: 9227465
```

```
In [54]: %%cython  
def fib_rec_cy(int n):  
    if n < 2:  
        return n  
    else:  
        return fib_rec_cy(n - 1) + fib_rec_cy(n - 2)
```

```
In [55]: %time fib_rec_cy(35)  
CPU times: user 751 ms, sys: 4.37 ms, total: 756 ms
```

```
Wall time: 755 ms
Out[55]: 9227465
```

Основной недостаток рекурсивных алгоритмов заключается в том, что промежуточные результаты пересчитываются, а не кешируются. Решить такую проблему позволяет декоратор, обеспечивающий кеширование промежуточных результатов, что позволяет увеличить скорость вычислений на два порядка.

```
In [56]: from functools import lru_cache as cache
```

```
In [57]: @cache(maxsize=None) ❶
def fib_rec_py2(n):
    if n < 2:
        return n
    else:
        return fib_rec_py2(n - 1) + fib_rec_py2(n - 2)
```

```
In [58]: %time fib_rec_py2(35) ❷
CPU times: user 64 µs, sys: 28 µs, total: 92 µs
Wall time: 98 µs
Out[58]: 9227465
```

```
In [59]: %time fib_rec_py2(80) ❷
CPU times: user 38 µs, sys: 8 µs, total: 46 µs
Wall time: 51 µs
Out[59]: 23416728348467685
```

❶ Кеширование промежуточных результатов...

❷ ...приводит к огромному ускорению вычислений.

## Итеративный алгоритм

Несмотря на то что алгоритм вычисления  $n$ -го числа Фибоначчи можно реализовать рекурсивным способом, это вовсе не единственное решение. Ниже представлена итеративная реализация, которая даже на чистом коде Python работает быстрее, чем рекурсивная реализация с кешированием. Еще большего ускорения можно добиться благодаря пакету Numba, но самый быстрый алгоритм реализуется с помощью пакета Cython.

```
In [60]: def fib_it_py(n):
    x, y = 0, 1
    for i in range(1, n + 1):
        x, y = y, x + y
    return x
```

```
In [61]: %time fib_it_py(80)
CPU times: user 19 µs, sys: 1e+03 ns, total: 20 µs
Wall time: 26 µs
Out[61]: 23416728348467685
```

```
In [62]: fib_it_nb = numba.jit(fib_it_py)
```

```
In [63]: %time fib_it_nb(80)
CPU times: user 57 ms, sys: 6.9 ms, total: 63.9 ms
Wall time: 62 ms
Out[63]: 23416728348467685
```

```
In [64]: %time fib_it_nb(80)
CPU times: user 7 µs, sys: 1 µs, total: 8 µs
Wall time: 12.2 µs
Out[64]: 23416728348467685
```

```
In [65]: %%cython
def fib_it_cy1(int n):
    cdef long i
    cdef long x = 0, y = 1
    for i in range(1, n + 1):
        x, y = y, x + y
    return x
```

```
In [66]: %time fib_it_cy1(80)
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 11 µs
Out[66]: 23416728348467685
```

Вы спросите, почему при такой скорости вычислений мы определяем только 80-е число последовательности, а не, например, 150-е? Оказывается, ограничение связано с доступными типами данных. Как описывалось в главе 3, Python способен работать с произвольно большими числами, но в случае компилируемых языков это не так. В частности, объект `double` позволяет хранить вещественные числа длиной 64 бита. Для работы с еще большими числами в пакете Cython предусмотрен специальный тип данных.

```
In [67]: %time
fn = fib_rec_py2(150) ❶
print(fn) ❶
9969216677189303386214405760200
```

```
CPU times: user 361 µs, sys: 115 µs, total: 476 µs
Wall time: 430 µs
```

```
In [68]: fn.bit_length() ❷
```

```
Out[68]: 103
```

```
In [69]: %%time
```

```
fn = fib_it_nb(150) ❸
```

```
print(fn) ❸
```

```
6792540214324356296
```

```
CPU times: user 270 µs, sys: 78 µs, total: 348 µs
```

```
Wall time: 297 µs
```

```
In [70]: fn.bit_length() ❹
```

```
Out[70]: 63
```

```
In [71]: %%time
```

```
fn = fib_it_cy1(150) ❸
```

```
print(fn) ❸
```

```
6792540214324356296
```

```
CPU times: user 255 µs, sys: 71 µs, total: 326 µs
```

```
Wall time: 279 µs
```

```
In [72]: fn.bit_length() ❹
```

```
Out[72]: 63
```

```
In [73]: %%cython
```

```
cdef extern from *:
```

```
    ctypedef int int128 '__int128_t' ❺
```

```
def fib_it_cy2(int n):
```

```
    cdef int128 i ❺
```

```
    cdef int128 x = 0, y = 1 ❺
```

```
    for i in range(1, n + 1):
```

```
        x, y = y, x + y
```

```
    return x
```

```
In [74]: %%time
```

```
fn = fib_it_cy2(150) ❻
```

```
print(fn) ❻
```

```
9969216677189303386214405760200
```

```
CPU times: user 280 µs, sys: 115 µs, total: 395 µs
```

```
Wall time: 328 µs
```

```
In [75]: fn.bit_length() ⑥
Out[75]: 103
```

- ❶ Чистый код Python работает достаточно быстро и возвращает корректный результат.
- ❷ Битовая длина возвращаемого целочисленного значения равна 103 ( $> 64$ ).
- ❸ Версии Numba и Cython работают быстрее, но возвращают неправильные результаты.
- ❹ Это связано с проблемой переполнения, поскольку для объектов `int` максимальная длина — 64 бита.
- ❺ Импорт специального 128-битного типа объектов `int`.
- ❻ Теперь функция `fib_it_cy2()`, написанная на Cython, не только работает быстрее, но и возвращает правильный результат.

## Число $\pi$

Последний из рассматриваемых в данном разделе алгоритмов — определение разрядов числа  $\pi$  по методу Монте-Карло<sup>2</sup>. Метод основан на формуле вычисления площади круга  $A$ , имеющей вид  $A = \pi r^2$ . Отсюда  $\pi = \frac{A}{r^2}$ . У круга с единичным радиусом  $\pi = A$ . Идея алгоритма заключается в моделировании случайных точек с координатами  $(x, y)$  в пространстве  $x, y \in [-1, 1]$ . Площадь квадрата, в который вписан круг единичного радиуса, равна 4, а площадь самого круга составляет дробную часть площади такого квадрата. Метод Монте-Карло позволяет выразить эту дробь через соотношение количества случайных точек, попадающих внутрь круга и квадрата. Реализация алгоритма приведена ниже (рис. 10.1).

```
In [76]: import random
import numpy as np
from pylab import mpl, plt
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

```
In [77]: rn = [(random.random() * 2 - 1, random.random() * 2 - 1)
for _ in range(500)]
```

---

<sup>2</sup>Примеры навеяны статьей на сайте Code Review Stack Exchange (<https://codereview.stackexchange.com/questions/69370/monte-carlo-pi-calculation>).



```
In [78]: rn = np.array(rn)
         rn[:5]
Out[78]: array([[ 0.45583018, -0.27676067],
                [-0.70120038,  0.15196888],
                [ 0.07224045,  0.90147321],
                [-0.17450337, -0.47660912],
                [ 0.94896746, -0.31511879]])
```

```
In [79]: fig = plt.figure(figsize=(7, 7))
         ax = fig.add_subplot(1, 1, 1)
         circ = plt.Circle((0, 0), radius=1, edgecolor='g',
                           lw=2.0, facecolor='None') ❶
         box = plt.Rectangle((-1, -1), 2, 2, edgecolor='b',
                              alpha=0.3) ❷
         ax.add_patch(circ) ❶
         ax.add_patch(box) ❷
         plt.plot(rn[:, 0], rn[:, 1], 'r.') ❸
         plt.ylim(-1.1, 1.1)
         plt.xlim(-1.1, 1.1)
```

- ❶ Рисуем круг единичного радиуса.
- ❷ Рисуем квадрат с длиной стороны, равной 2.
- ❸ Отображение случайных точек, имеющих нормальное распределение.

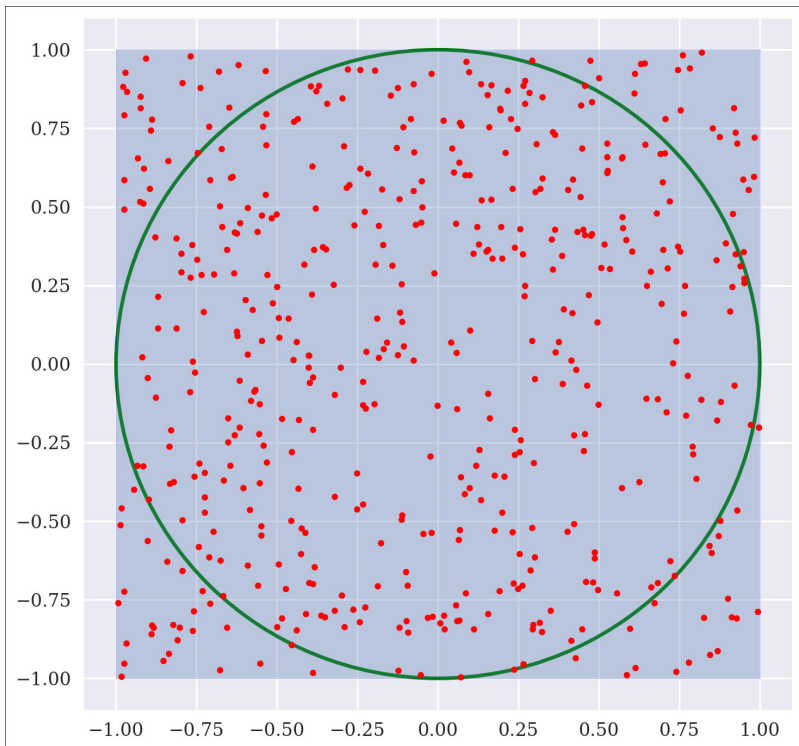
Применение пакета NumPy позволяет получить компактную реализацию алгоритма, но требует больших ресурсов памяти. Общее время выполнения с учетом параметризации составляет примерно одну секунду.

```
In [80]: n = int(1e7)
```

```
In [81]: %time rn = np.random.random((n, 2)) * 2 - 1
CPU times: user 450 ms, sys: 87.9 ms, total: 538 ms
Wall time: 573 ms
```

```
In [82]: rn.nbytes
Out[82]: 160000000
```

```
In [83]: %time distance = np.sqrt((rn ** 2).sum(axis=1)) ❶
         distance[:8].round(3)
CPU times: user 537 ms, sys: 198 ms, total: 736 ms
Wall time: 651 ms
```



*Рис. 10.1. Случайные точки с нормальным распределением, отложенные в квадрате с длиной стороны 2, в который вписан круг единичного радиуса*

```
Out[83]: array([1.181, 1.061, 0.669, 1.206, 0.799, 0.579,
               0.694, 0.941])
```

```
In [84]: %time frac = (distance <= 1.0).sum() / len(distance) ❷
CPU times: user 47.9 ms, sys: 6.77 ms, total: 54.7 ms
Wall time: 28 ms
```

```
In [85]: pi_mcs = frac * 4 ❸
pi_mcs ❸
```

```
Out[85]: 3.1413396
```

- ❶ Расстояние до начала координат (евклидова норма).
- ❷ Отношение количества точек, попадающих в круг, к общему числу точек.
- ❸ Выражение числа  $\pi$  с учетом того, что площадь квадрата равна 4.

Функция `mcs_pi_py()` реализует метод Монте-Карло на чистом Python с использованием цикла `for`, не требуя больших ресурсов памяти. В данной реализации случайные числа не масштабируются. Производительность получается меньше, чем в версии NumPy, но благодаря пакету Numba скорость вычислений удастся повысить.

```
In [86]: def mcs_pi_py(n):
        circle = 0
        for _ in range(n):
            x, y = random.random(), random.random()
            if (x ** 2 + y ** 2) ** 0.5 <= 1:
                circle += 1
        return (4 * circle) / n
```

```
In [87]: %time mcs_pi_py(n)
CPU times: user 5.47 s, sys: 23 ms, total: 5.49 s
Wall time: 5.43 s
Out[87]: 3.1418964
```

```
In [88]: mcs_pi_nb = numba.jit(mcs_pi_py)
```

```
In [89]: %time mcs_pi_nb(n)
CPU times: user 319 ms, sys: 6.36 ms, total: 326 ms
Wall time: 326 ms
Out[89]: 3.1422012
```

```
In [90]: %time mcs_pi_nb(n)
CPU times: user 284 ms, sys: 3.92 ms, total: 288 ms
Wall time: 291 ms
Out[90]: 3.142066
```

Версия Cython, в которой используется только статическая типизация, работает не намного быстрее, чем версия на чистом Python. Но если применить генератор случайных чисел языка C, то вычисления существенно ускорятся.

```
In [91]: %%cython -a
import random
def mcs_pi_cy1(int n):
    cdef int i, circle = 0
    cdef float x, y
    for i in range(n):
        x, y = random.random(), random.random()
        if (x ** 2 + y ** 2) ** 0.5 <= 1:
            circle += 1
```

```

        return (4 * circle) / n
Out[91]: <IPython.core.display.HTML object>

In [92]: %time mcs_pi_cy1(n)
CPU times: user 1.15 s, sys: 8.24 ms, total: 1.16 s
Wall time: 1.16 s
Out[92]: 3.1417132

In [93]: %%cython -a
from libc.stdlib cimport rand
cdef extern from 'limits.h':
    int INT_MAX
def mcs_pi_cy2(int n):
    cdef int i, circle = 0
    cdef float x, y
    for i in range(n):
        x, y = rand() / INT_MAX, rand() / INT_MAX
        if (x ** 2 + y ** 2) ** 0.5 <= 1:
            circle += 1
    return (4 * circle) / n
Out[93]: <IPython.core.display.HTML object>

In [94]: %time mcs_pi_cy2(n)
CPU times: user 170 ms, sys: 1.45 ms, total: 172 ms
Wall time: 172 ms
Out[94]: 3.1419388

```



### Типы алгоритмов

Алгоритмы, рассмотренные в этом разделе, не связаны напрямую с обработкой финансовых данных. Зато они достаточно простые и позволяют понять, какие проблемы могут возникать в финансовых приложениях.

## Биномиальные деревья

Биномиальная модель оценки опционов, впервые предложенная Коксом, Россом и Рубинштейном (1979), — это популярный численный метод прогнозирования стоимости опционов, в котором эволюция цены базового актива выражается через биномиальное дерево всех возможных цен. Как и в модели Блэка — Шоулза — Мертона (1973), здесь есть *рисковый актив* (индекс или акция), и *безрисковый актив* (облигация). Интервал времени от текущего

момента до экспирации опциона разбивается на отрезки равной длины ( $\Delta t$ ). Если в момент времени  $s$  уровень индекса равен  $S_s$ , то в момент времени  $t = s + \Delta t$  его можно выразить как  $S_t = S_s \cdot m$ , где  $m$  — случайное значение из интервала  $\{u, d\}$ ,  $0 < d < e^{r\Delta t} < u = e^{\sigma\sqrt{\Delta t}}$ ,  $u = \frac{1}{d}$ ,  $r$  — постоянная безрисковая краткосрочная ставка.

## Python

Ниже приведен код Python, в котором создается биномиальное дерево на основе фиксированных числовых параметров модели.

```
In [95]: import math
```

```
In [96]: S0 = 36. ❶
          T = 1.0 ❷
          r = 0.06 ❸
          sigma = 0.2 ❹
```

```
In [97]: def simulate_tree(M):
          dt = T / M ❺
          u = math.exp(sigma * math.sqrt(dt)) ❻
          d = 1 / u ❼
          S = np.zeros((M + 1, M + 1))
          S[0, 0] = S0
          z = 1
          for t in range(1, M + 1):
              for i in range(z):
                  S[i, t] = S[i, t-1] * u
                  S[i+1, t] = S[i, t-1] * d
              z += 1
          return S
```

- ❶ Начальная стоимость рискованного актива.
- ❷ Временной горизонт, для моделирования биномиального дерева.
- ❸ Постоянная краткосрочная процентная ставка.
- ❹ Фиксированный коэффициент волатильности.
- ❺ Длительность временных интервалов (шагов).
- ❻ Коэффициенты движения цены вверх или вниз.

В отличие от обычных графов, здесь движение цены вверх представляется в объекте `ndarray` как боковое движение, которое приводит к существенному уменьшению его размера.

```
In [98]: np.set_printoptions(formatter={'float':  
                                         lambda x: '%6.2f' % x})
```

```
In [99]: simulate_tree(4) ❶  
Out[99]: array([[ 36.00,  39.79,  43.97,  48.59,  53.71],  
                [  0.00,  32.57,  36.00,  39.79,  43.97],  
                [  0.00,   0.00,  29.47,  32.57,  36.00],  
                [  0.00,   0.00,   0.00,  26.67,  29.47],  
                [  0.00,   0.00,   0.00,   0.00,  24.13]])
```

```
In [100]: %time simulate_tree(500) ❷  
CPU times: user 148 ms, sys: 4.49 ms, total: 152 ms  
Wall time: 154 ms  
Out[100]: array([[ 36.00,  36.32,  36.65, ..., 3095.69, 3123.50, 3151.57],  
                [  0.00,  35.68,  36.00, ..., 3040.81, 3068.13, 3095.69],  
                [  0.00,   0.00,  35.36, ..., 2986.89, 3013.73, 3040.81],  
                ...,  
                [  0.00,   0.00,   0.00, ...,   0.42,   0.42,   0.43],  
                [  0.00,   0.00,   0.00, ...,   0.00,   0.41,   0.42],  
                [  0.00,   0.00,   0.00, ...,   0.00,   0.00,   0.41]])
```

- ❶ Дерево, содержащее 4 временных интервала.
- ❷ Дерево, содержащее 500 временных интервалов.

## NumPy

Биномиальное дерево можно реализовать с помощью полностью векторизованного кода NumPy.

```
In [101]: M = 4  
  
In [102]: up = np.arange(M + 1)  
          up = np.resize(up, (M + 1, M + 1)) ❶  
          up  
Out[102]: array([[0, 1, 2, 3, 4],  
                [0, 1, 2, 3, 4],  
                [0, 1, 2, 3, 4],  
                [0, 1, 2, 3, 4],  
                [0, 1, 2, 3, 4]])
```

```
In [103]: down = up.T * 2 ❷
```

```
down
```

```
Out[103]: array([[0, 0, 0, 0, 0],
                 [2, 2, 2, 2, 2],
                 [4, 4, 4, 4, 4],
                 [6, 6, 6, 6, 6],
                 [8, 8, 8, 8, 8]])
```

```
In [104]: up - down ❸
```

```
Out[104]: array([[ 0,  1,  2,  3,  4],
                 [-2, -1,  0,  1,  2],
                 [-4, -3, -2, -1,  0],
                 [-6, -5, -4, -3, -2],
                 [-8, -7, -6, -5, -4]])
```

```
In [105]: dt = T / M
```

```
In [106]: S0 * np.exp(sigma * math.sqrt(dt) * (up - down)) ❹
```

```
Out[106]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                 [ 29.47, 32.57, 36.00, 39.79, 43.97],
                 [ 24.13, 26.67, 29.47, 32.57, 36.00],
                 [ 19.76, 21.84, 24.13, 26.67, 29.47],
                 [ 16.18, 17.88, 19.76, 21.84, 24.13]])
```

- ❶ Объект `ndarray`, описывающий общее движение цены *вверх*.
- ❷ Объект `ndarray`, описывающий общее движение цены *вниз*.
- ❸ Объект `ndarray`, описывающий *остаточное* движение цены вверх (положительное значение) и вниз (отрицательное значение).
- ❹ Биномиальное дерево (правый верхний треугольник значений) для четырех временных интервалов.

Как видите, в случае NumPy код получается компактнее. Но важнее всего то, что векторизация приводит к ускорению вычислений на порядок, не требуя при этом дополнительной памяти.

```
In [107]: def simulate_tree_np(M):
           dt = T / M
           up = np.arange(M + 1)
           up = np.resize(up, (M + 1, M + 1))
           down = up.transpose() * 2
```

```
S = S0 * np.exp(sigma * math.sqrt(dt) * (up - down))
return S
```

```
In [108]: simulate_tree_np(4)
```

```
Out[108]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
 [ 29.47, 32.57, 36.00, 39.79, 43.97],
 [ 24.13, 26.67, 29.47, 32.57, 36.00],
 [ 19.76, 21.84, 24.13, 26.67, 29.47],
 [ 16.18, 17.88, 19.76, 21.84, 24.13]])
```

```
In [109]: %time simulate_tree_np(500)
```

```
CPU times: user 8.72 ms, sys: 7.07 ms, total: 15.8 ms
```

```
Wall time: 12.9 ms
```

```
Out[109]: array([[ 36.00, 36.32, 36.65, ..., 3095.69, 3123.50, 3151.57],
 [ 35.36, 35.68, 36.00, ..., 3040.81, 3068.13, 3095.69],
 [ 34.73, 35.05, 35.36, ..., 2986.89, 3013.73, 3040.81],
 ...,
 [ 0.00, 0.00, 0.00, ..., 0.42, 0.42, 0.43],
 [ 0.00, 0.00, 0.00, ..., 0.41, 0.41, 0.42],
 [ 0.00, 0.00, 0.00, ..., 0.40, 0.41, 0.41]])
```

## Numba

Наш финансовый алгоритм должен хорошо оптимизироваться за счет динамической компиляции, реализуемой пакетом Numba. Как показывает следующий пример, производительность увеличивается на порядок по сравнению с версией NumPy. Таким образом, если сравнивать с начальной версией, написанной на чистом Python, то Numba позволяет ускорить вычисления на несколько порядков.

```
In [110]: simulate_tree_nb = numba.jit(simulate_tree)
```

```
In [111]: simulate_tree_nb(4)
```

```
Out[111]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
 [ 0.00, 32.57, 36.00, 39.79, 43.97],
 [ 0.00, 0.00, 29.47, 32.57, 36.00],
 [ 0.00, 0.00, 0.00, 26.67, 29.47],
 [ 0.00, 0.00, 0.00, 0.00, 24.13]])
```

```
In [112]: %time simulate_tree_nb(500)
```

```
CPU times: user 425 µs, sys: 193 µs, total: 618 µs
```

```
Wall time: 625 µs
```



```
Out[112]: array([[ 36.00, 36.32, 36.65, ..., 3095.69, 3123.50, 3151.57],
 [  0.00, 35.68, 36.00, ..., 3040.81, 3068.13, 3095.69],
 [  0.00,  0.00, 35.36, ..., 2986.89, 3013.73, 3040.81],
 ...,
 [  0.00,  0.00,  0.00, ...,  0.42,  0.42,  0.43],
 [  0.00,  0.00,  0.00, ...,  0.00,  0.41,  0.42],
 [  0.00,  0.00,  0.00, ...,  0.00,  0.00,  0.41]])
```

```
In [113]: %timeit simulate_tree_nb(500)
559 µs ± 46.1 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

## Cython

Как и ранее, чтобы получить заметный выигрыш в производительности при использовании Cython, необходимо дополнительно адаптировать код. В приведенной ниже реализации применяется статическая типизация и импортируются специальные библиотеки, повышающие производительность в сравнении с функциями из стандартной библиотеки Python.

```
In [114]: %%cython -a
import numpy as np
import cython
from libc.math cimport exp, sqrt
cdef float S0 = 36.
cdef float T = 1.0
cdef float r = 0.06
cdef float sigma = 0.2
def simulate_tree_cy(int M):
    cdef int z, t, i
    cdef float dt, u, d
    cdef float[:, :] S = np.zeros((M + 1, M + 1),
                                   dtype=np.float32) ❶

    dt = T / M
    u = exp(sigma * sqrt(dt))
    d = 1 / u
    S[0, 0] = S0
    z = 1
    for t in range(1, M + 1):
        for i in range(z):
            S[i, t] = S[i, t-1] * u
            S[i+1, t] = S[i, t-1] * d
```

```

        z += 1
    return np.array(S)
Out[114]: <IPython.core.display.HTML object>

```

- ❶ Объявление объекта `ndarray` в виде массива C; критически важное решение для увеличения производительности.

Применение пакета Cython позволяет сократить время выполнения кода еще на 30% по сравнению с версией Numba.

```

In [115]: simulate_tree_cy(4)
Out[115]: array([[ 36.00, 39.79, 43.97, 48.59, 53.71],
                 [  0.00, 32.57, 36.00, 39.79, 43.97],
                 [  0.00,  0.00, 29.47, 32.57, 36.00],
                 [  0.00,  0.00,  0.00, 26.67, 29.47],
                 [  0.00,  0.00,  0.00,  0.00, 24.13]],
                dtype=float32)

In [116]: %time simulate_tree_cy(500)
CPU times: user 2.21 ms, sys: 1.89 ms, total: 4.1 ms
Wall time: 2.45 ms

Out[116]: array([[ 36.00, 36.32, 36.65, ..., 3095.77, 3123.59, 3151.65],
                 [  0.00, 35.68, 36.00, ..., 3040.89, 3068.21, 3095.77],
                 [  0.00,  0.00, 35.36, ..., 2986.97, 3013.81, 3040.89],
                 ...,
                 [  0.00,  0.00,  0.00, ...,  0.42,  0.42,  0.43],
                 [  0.00,  0.00,  0.00, ...,  0.00,  0.41,  0.42],
                 [  0.00,  0.00,  0.00, ...,  0.00,  0.00,  0.41]],
                dtype=float32)

In [117]: %timeit S = simulate_tree_cy(500)
363 µs ± 29.5 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)

```

## Метод Монте-Карло

Метод Монте-Карло — незаменимый помощник в финансовых расчетах. Он был известен задолго до появления современных компьютеров. Банки и другие финансовые учреждения применяют его, в частности, для прогнозирования цен и управления рисками. Это один из самых гибких и мощных численных методов финансовой математики. В то же время он, как правило,

самый требовательный к вычислительным ресурсам. Именно поэтому в течение долгого времени Python считался неподходящим выбором для реализации алгоритмов, основанных на методе Монте-Карло, по крайней мере в прикладных решениях.

В этом разделе мы будем моделировать геометрическое броуновское движение — простой стохастический процесс, широко применяемый для изучения эволюции курсов акции или биржевых индексов. На основе этого процесса, в частности, была построена модель ценообразования опционов Блэка — Шоулза — Мертона, согласно которой цена базового актива рассчитывается по стохастическому дифференциальному уравнению 10.1, где  $S_t$  — цена базового актива в момент времени  $t$ ,  $r$  — постоянная безрисковая краткосрочная ставка,  $s$  — постоянная моментальная волатильность, а  $Z_t$  — броуновское движение.

**Уравнение 10.1. Стохастическое дифференциальное уравнение Блэка — Шоулза — Мертона (геометрическое броуновское движение)**

$$dS_t = rS_t dt + \sigma S_t dZ_t.$$

Такое уравнение можно решить методом Эйлера, разбив горизонт прогнозирования на равные интервалы и воспользовавшись уравнением 10.2, где  $z$  — случайное число из стандартного нормального распределения. В случае  $M$  временных интервалов длина интервала рассчитывается как  $\Delta t \equiv \frac{T}{M}$ , где  $T$  — горизонт прогнозирования (например, срок исполнения опциона).

**Уравнение 10.2. Рекуррентное уравнение Блэка — Шоулза — Мертона (метод Эйлера)**

$$S_t = S_{t-\Delta t} \exp \left( \left( r - \frac{\sigma^2}{2} \right) \Delta t + \sigma \sqrt{\Delta t} z \right).$$

Цена европейского колл-опциона моделируется по методу Монте-Карло с помощью уравнения 10.3, где  $S_T(i)$  —  $i$ -я моделируемая стоимость базового актива со сроком исполнения  $T$  и общим количеством траекторий  $I$  ( $i = 1, 2, \dots, I$ ).

**Уравнение 10.3. Оценка стоимости европейского колл-опциона по методу Монте-Карло**

$$C_0 = e^{-rT} \frac{1}{I} \sum_I \max(S_T(t) - K, 0).$$

## Python

Вначале рассмотрим гибридную версию функции `mcs_simulation_py()`, которая реализует моделирование по методу Монте-Карло в соответствии с уравнением 10.2. Гибридность обусловлена использованием объектов `ndarray` в цикле. Как было показано ранее, это делает оправданной динамическую компиляцию кода с помощью пакета `Numba`. Время выполнения данной функции послужит нам эталоном, с которым будут сравниваться последующие реализации. На основе результатов моделирования вычисляется стоимость европейского пут-опциона.

```
In [118]: M = 100 ❶  
          I = 50000 ❷
```

```
In [119]: def mcs_simulation_py(p):  
           M, I = p  
           dt = T / M  
           S = np.zeros((M + 1, I))  
           S[0] = S0  
           rn = np.random.standard_normal(S.shape) ❸  
           for t in range(1, M + 1): ❹  
               for i in range(I): ❺  
                   S[t, i] = S[t-1, i] * math.exp((r - sigma ** \  
                       2 / 2) * dt + sigma * math.sqrt(dt) * \  
                       rn[t, i]) ❻  
           return S
```

```
In [120]: %time S = mcs_simulation_py((M, I))  
CPU times: user 5.55 s, sys: 52.9 ms, total: 5.6 s  
Wall time: 5.62 s
```

```
In [121]: S[-1].mean() ❼  
Out[121]: 38.22291254503985
```

```
In [122]: S0 * math.exp(r * T) ❽  
Out[122]: 38.22611567563295
```

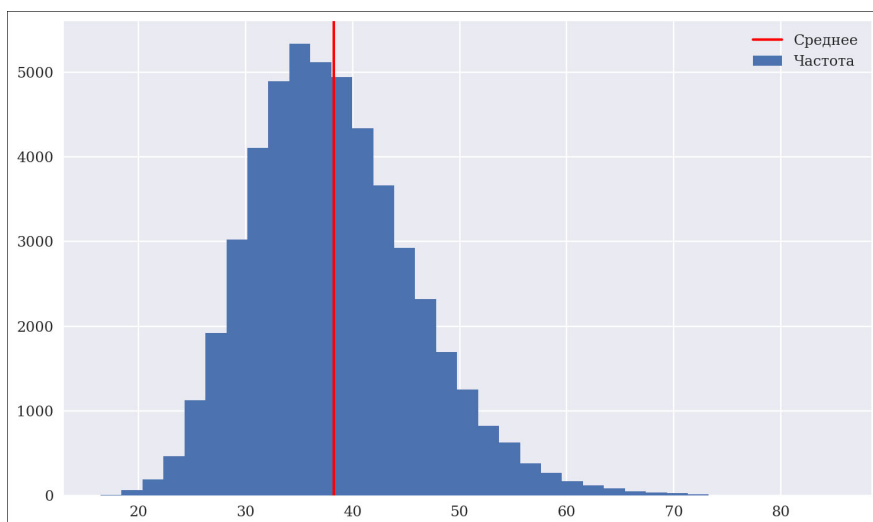
```
In [123]: K = 40. ❹
```

```
In [124]: C0 = math.exp(-r * T) * np.maximum(K - S[-1], 0).mean() ❺
```

```
In [125]: C0 ❻  
Out[125]: 3.860545188088036
```

- ❶ Количество интервалов дискретизации.
- ❷ Количество моделируемых траекторий.
- ❸ Случайные числа, векторизованно генерируемые за один проход.
- ❹ Вложенные циклы, в которых реализуется моделирование по методу Эйлера.
- ❺ Средняя стоимость на конец временного периода.
- ❻ Теоретически ожидаемая стоимость на конец временного периода.
- ❼ Страйк-цена европейского пут-опциона.
- ❽ Оценка стоимости опциона по методу Монте-Карло.

На рис. 10.2 показана гистограмма полученных в процессе моделирования значений на конец каждого временного периода (срока исполнения европейского пут-опциона).



*Рис. 10.2. Частотное распределение моделированной стоимости опциона на конец каждого периода*

## NumPy

NumPy-версия функции `mcs_simulation_np()` не сильно отличается от базового варианта. В ней по-прежнему имеется один цикл Python, обеспечива-

ющий пошаговую обработку временных интервалов. Второй цикл “поглощается” векторизованным кодом, что позволяет ускорить вычисления в 20 раз.

```
In [127]: def mcs_simulation_np(p):  
    M, I = p  
    dt = T / M  
    S = np.zeros((M + 1, I))  
    S[0] = S0  
    rn = np.random.standard_normal(S.shape)  
    for t in range(1, M + 1): ❶  
        S[t] = S[t-1] * np.exp((r - sigma ** 2 / 2) *  
                                dt + sigma * math.sqrt(dt) * rn[t]) ❷  
    return S
```

```
In [128]: %time S = mcs_simulation_np((M, I))  
CPU times: user 252 ms, sys: 32.9 ms, total: 285 ms  
Wall time: 252 ms
```

```
In [129]: S[-1].mean()  
Out[129]: 38.235136032258595
```

```
In [130]: %timeit S = mcs_simulation_np((M, I))  
202 ms ± 27.7 ms per loop (mean ± std. dev. of 7 runs,  
1 loop each)
```

- ❶ Пошаговая обработка временных интервалов.
- ❷ Вычисления по методу Эйлера выполняются с помощью векторизованных инструментов NumPy, что позволяет обрабатывать все траектории за один проход.

## Numba

При реализации подобного рода алгоритмов имеет смысл применять пакет Numba, что позволяет добиться повышения производительности. Как показано ниже, Numba-версия функции `mcs_simulation_nb()` работает чуть быстрее, чем NumPy-версия.

```
In [131]: mcs_simulation_nb = numba.jit(mcs_simulation_py)
```

```
In [132]: %time S = mcs_simulation_nb((M, I)) ❶  
CPU times: user 673 ms, sys: 36.7 ms, total: 709 ms  
Wall time: 764 ms
```

```

In [133]: %time S = mcs_simulation_nb((M, I)) ❷
          CPU times: user 239 ms, sys: 20.8 ms, total: 259 ms
          Wall time: 265 ms

In [134]: S[-1].mean()
Out[134]: 38.22350694016539

In [135]: C0 = math.exp(-r * T) * np.maximum(K - S[-1], 0).mean()

In [136]: C0
Out[136]: 3.8303077438193833

In [137]: %timeit S = mcs_simulation_nb((M, I)) ❷
          248 ms ± 20.6 ms per loop (mean ± std. dev. of 7 runs,
          1 loop each)

```

- ❶ При первом вызове возникают накладные расходы, связанные с компиляцией.
- ❷ Второй вызов выполняется значительно быстрее.

## Cython

При использовании пакета Cython необходимо вносить изменения в код. Однако значительного ускорения вычислений здесь не происходит. Функция `mcs_simulation_cy()` работает даже чуть медленнее, чем версии NumPy и Numba. Это связано с необходимостью преобразования результатов моделирования в объект `ndarray`, что отнимает время.

```

In [138]: %%cython
          import numpy as np
          cimport numpy as np
          cimport cython
          from libc.math cimport exp, sqrt
          cdef float S0 = 36.
          cdef float T = 1.0
          cdef float r = 0.06
          cdef float sigma = 0.2
          @cython.boundscheck(False)
          @cython.wraparound(False)
          def mcs_simulation_cy(p):
              cdef int M, I
              M, I = p

```

```

cdef int t, i
cdef float dt = T / M
cdef double[:, :] S = np.zeros((M + 1, I))
cdef double[:, :] rn = \
    np.random.standard_normal((M + 1, I))
S[0] = S0
for t in range(1, M + 1):
    for i in range(I):
        S[t, i] = S[t-1, i] * \
            exp((r - sigma ** 2 / 2) * \
                dt + sigma * sqrt(dt) * rn[t, i])
return np.array(S)

```

```

In [139]: %time S = mcs_simulation_cy((M, I))
CPU times: user 237 ms, sys: 65.2 ms, total: 302 ms
Wall time: 271 ms

```

```

In [140]: S[-1].mean()
Out[140]: 38.241735841791574

```

```

In [141]: %timeit S = mcs_simulation_cy((M, I))
221 ms ± 9.26 ms per loop (mean ± std. dev. of 7 runs,
1 loop each)

```

## Параллельные вычисления

Моделирование по методу Монте-Карло — это задача, которая хорошо поддается параллельной обработке. Можно, например, распараллелить вычисление 100 000 траекторий на 10 процессов, каждый из которых будет моделировать 10 000 траекторий. Другой вариант — моделировать 100 000 траекторий разными процессами, каждый из которых соответствует отдельному финансовому инструменту. В этом разделе мы рассмотрим первый сценарий.

Мы снова воспользуемся модулем `multiprocessing`. В следующем коде общее количество траекторий  $I$  разбивается на несколько пакетов размером  $I/p$  каждый, где  $p > 0$ . По завершении отдельных процессов результаты заносятся в объект `ndarray` с помощью метода `np.hstack()`. Такой подход можно применить к любой из рассмотренных выше реализаций алгоритма. В данном случае выбранные параметры параллелизации не приводят к ускорению вычислений.



```
In [142]: import multiprocessing as mp
```

```
In [143]: pool = mp.Pool(processes=4) ❶
```

```
In [144]: p = 20 ❷
```

```
In [145]: %timeit S = np.hstack(pool.map(mcs_simulation_np,  
                                         p * [(M, int(I / p))]))  
288 ms ± 10.2 ms per loop (mean ± std. dev. of 7 runs,  
1 loop each)
```

```
In [146]: %timeit S = np.hstack(pool.map(mcs_simulation_nb,  
                                         p * [(M, int(I / p))]))  
258 ms ± 8.69 ms per loop (mean ± std. dev. of 7 runs,  
1 loop each)
```

```
In [147]: %timeit S = np.hstack(pool.map(mcs_simulation_cy,  
                                         p * [(M, int(I / p))]))  
274 ms ± 11.9 ms per loop (mean ± std. dev. of 7 runs,  
1 loop each)
```

- ❶ Объект `Pool`, предназначенный для распараллеливания вычислений.
- ❷ Количество пакетов, на которые разбивается блок моделируемых траекторий.



### Стратегии параллельной обработки

В финансовых вычислениях используется много алгоритмов, допускающих параллельную обработку. В некоторых случаях существует даже несколько стратегий распараллеливания кода. Метод Монте-Карло — прекрасный пример алгоритма, в котором можно распределить моделирование между несколькими процессами или же выполнять разные процессы моделирования на одном или нескольких компьютерах.

## Рекурсивный алгоритм библиотеки pandas

В этом разделе рассматривается важная тема, имеющая огромное значение в финансовых приложениях: реализация рекурсивных функций для обработки финансовых временных рядов, хранящихся в объекте `DataFrame` библиотеки `pandas`. Несмотря на то что библиотека `pandas` позволяет выполнять

векторизованные операции с объектами `DataFrame`, некоторые алгоритмы слишком сложно или даже невозможно векторизовать, и в результате финансовому аналитику приходится обрабатывать такие объекты в медленных циклах Python. В следующем примере мы реализуем простой алгоритм вычисления экспоненциально взвешенного скользящего среднего (Exponentially Weighted Moving Average — EWMA).

Значение EWMA для финансового временного ряда  $S_t$ ,  $t \in \{0, \dots, T\}$ , вычисляется согласно уравнению 10.4.

*Уравнение 10.4. Экспоненциально взвешенное скользящее среднее*

$$\begin{aligned}EWMA_0 &= S_0, \\EWMA_t &= \alpha S_t + (1 - \alpha)EWMA_{t-1}, t \in \{1, \dots, T\}.\end{aligned}$$

Несмотря на простоту реализации полученный код может работать достаточно медленно.

## Python

Вначале мы рассмотрим версию Python, в которой циклически обрабатываются значения `DatetimeIndex` объекта `DataFrame`, хранящего временной ряд отдельного финансового инструмента (см. главу 8). На рис. 10.3 визуализируется исходный временной ряд и временной ряд EWMA.

```
In [148]: import pandas as pd
```

```
In [149]: sym = 'SPY'
```

```
In [150]: data = pd.DataFrame(pd.read_csv( \
    '../..source/tr_eikon_eod_data.csv', index_col=0, \
    parse_dates=True)[sym]).dropna()
```

```
In [151]: alpha = 0.25
```

```
In [152]: data['EWMA'] = data[sym] ❶
```

```
In [153]: %%time
for t in zip(data.index, data.index[1:]):
    data.loc[t[1], 'EWMA'] = (alpha * data.loc[t[1], sym] + \
        (1 - alpha) * data.loc[t[0], 'EWMA']) ❷
CPU times: user 588 ms, sys: 16.4 ms, total: 605 ms
Wall time: 591 ms
```

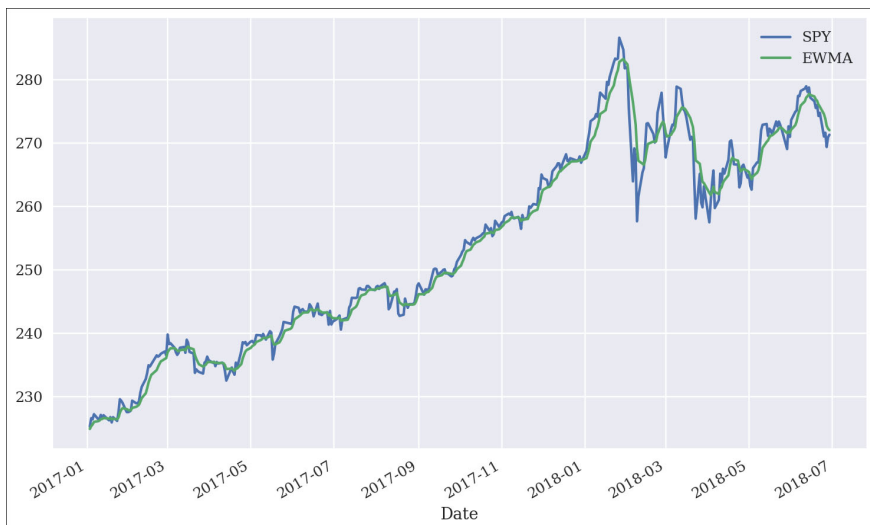
```
In [154]: data.head()
```

```
Out[154]:
```

	SPY	EWMA
<b>Date</b>		
<b>2010-01-04</b>	113.33	113.330000
<b>2010-01-05</b>	113.63	113.405000
<b>2010-01-06</b>	113.71	113.481250
<b>2010-01-07</b>	114.19	113.658438
<b>2010-01-08</b>	114.57	113.886328

```
In [155]: data[data.index > '2017-1-1'].plot(figsize=(10, 6));
```

- ❶ Инициализация столбца EWMA.
- ❷ Реализация алгоритма с помощью цикла Python.



*Рис. 10.3. Финансовый временной ряд и экспоненциально взвешенное скользящее среднее (EWMA)*

Теперь рассмотрим более общую функцию `ewma.py()`, которую можно применить к отдельному столбцу или исходному временному ряду, представленному в виде объекта `pandas`.

```
In [156]: def ewma_py(x, alpha):
            y = np.zeros_like(x)
            y[0] = x[0]
            for i in range(1, len(x)):
                y[i] = alpha * x[i] + (1-alpha) * y[i-1]
            return y
```

```
In [157]: %time data['EWMA_PY'] = ewma_py(data[sym], alpha) ❶
CPU times: user 33.1 ms, sys: 1.22 ms, total: 34.3 ms
Wall time: 33.9 ms
```

```
In [158]: %time data['EWMA_PY'] = ewma_py(data[sym].values, alpha) ❷
CPU times: user 1.61 ms, sys: 44 µs, total: 1.65 ms
Wall time: 1.62 ms
```

- ❶ Применение функции непосредственно к объекту `Series` (т.е. к столбцу).
- ❷ Применение функции к объекту `ndarray`, содержащему исходный временной ряд.

В результате мы получаем ускорение от 20 до более чем 100 раз.

## Numba

Дальнейшего ускорения можно добиться с помощью пакета Numba, который весьма эффективен для такого типа алгоритмов. И действительно, если применить функцию `ewma_nb()` к объекту `ndarray`, то можно сократить время вычислений еще на порядок.

```
In [159]: ewma_nb = numba.jit(ewma_py)
```

```
In [160]: %time data['EWMA_NB'] = ewma_nb(data[sym], alpha) ❶
CPU times: user 269 ms, sys: 11.4 ms, total: 280 ms
Wall time: 294 ms
```

```
In [161]: %timeit data['EWMA_NB'] = ewma_nb(data[sym], alpha) ❶
30.9 ms ± 1.21 ms per loop (mean ± std. dev. of 7 runs,
10 loops each)
```

```
In [162]: %time data['EWMA_NB'] = ewma_nb(data[sym].values, alpha) ❷
CPU times: user 94.1 ms, sys: 3.78 ms, total: 97.9 ms
Wall time: 97.6 ms
```

```
In [163]: %timeit data['EWMA_NB'] = ewma_nb(data[sym].values, alpha) ❷
134 µs ± 12.5 µs per loop (mean ± std. dev. of 7 runs,
10000 loops each)
```

- ❶ Применение функции непосредственно к объекту `Series` (т.е. к столбцу).
- ❷ Применение функции к объекту `ndarray`, содержащему исходный временной ряд.

## Cython

Функция `ewma_cy()`, написанная на Cython, тоже позволяет существенно ускорить вычисления, хотя и не настолько, как в предыдущем случае.

```
In [164]: %%cython
import numpy as np
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
def ewma_cy(double[:] x, float alpha):
    cdef int i
    cdef double[:] y = np.empty_like(x)
    y[0] = x[0]
    for i in range(1, len(x)):
        y[i] = alpha * x[i] + (1 - alpha) * y[i - 1]
    return y

In [165]: %time data['EWMA_CY'] = ewma_cy(data[sym].values, alpha)
CPU times: user 2.98 ms, sys: 1.41 ms, total: 4.4 ms
Wall time: 5.96 ms

In [166]: %timeit data['EWMA_CY'] = ewma_cy(data[sym].values, alpha)
1.29 ms ± 194 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

Этот пример наглядно показывает, что зачастую существует несколько вариантов реализации (нестандартных) алгоритмов. Результаты будут одними и теми же, но производительность решений оказывается разной. Время выполнения кода в данном случае варьируется от 0,1 до 500 мс (разница более чем в 5000 раз).



### Наилучший или первый из лучших

Как правило, алгоритмы достаточно легко транслируются в код Python. Но вместе с тем можно легко получить такую реализацию алгоритма, которая будет работать неприемлемо медленно в сравнении с другими возможными вариантами. Для задач интерактивного финансового анализа вполне достаточно выбрать *первое из лучших* решений, т.е. такое, которое обеспечит ускорение вычислений, но не обязательно окажется самым производительным или эффективным с точки зрения использования памяти. В те же время в коммерческих приложениях необходимо стремиться к получению *наилучшего* решения, пусть даже для этого придется проводить дополнительный анализ и формальное тестирование.

# Резюме

Экосистема Python предлагает различные способы повышения производительности кода.

## *Идиомы и парадигмы*

Очень часто при решении определенных задач отдельные идиомы и парадигмы Python приводят к получению более производительного кода. В частности, парадигма векторизации вычислений не только позволяет писать более компактный код, но и обеспечивает существенное ускорение вычислений (иногда за счет потребления дополнительной оперативной памяти).

## *Программные пакеты*

В Python имеется множество пакетов, предназначенных для решения различных задач. Использование соответствующего пакета часто позволяет получить более производительное решение. Хорошим примером могут служить пакеты NumPy (содержит класс `ndarray`) и pandas (содержит класс `DataFrame`).

## *Компиляция кода*

Numba и Cython — мощные пакеты, позволяющие повышать производительность финансовых алгоритмов за счет динамической и статической компиляции кода Python.

## *Параллельные вычисления*

Существуют специальные пакеты, в частности `multiprocessing`, позволяющие распараллелить выполнение кода Python. В этой главе мы рассматривали примеры параллельной обработки на одном компьютере, но экосистема Python поддерживает и технологии кластерной параллелизации.

Основное преимущество рассмотренных в этой главе подходов к повышению производительности кода — простота реализации. Они не требуют приложения больших усилий. Другими словами, благодаря имеющимся пакетам можно получить высокопроизводительный код без дополнительных затрат.

## Дополнительные ресурсы

Для всех рассмотренных в этой главе пакетов имеется много полезных ресурсов в Интернете.

- Пакет Cython и соответствующий проект компилятора описаны на официальном сайте (<http://cython.org>);
- Документацию к модулю `multiprocessing` можно найти на сайте <https://docs.python.org/3/library/multiprocessing.html>;
- Информацию о пакете Numba можно найти на сайтах <http://github.com/numba/numba> и <https://numba.pydata.org>.

Также рекомендуем следующие книги.

- Gorelick, Micha, and Ian Ozsvald. *High Performance Python* (2014, O'Reilly).
- Smith, Kurt. *Cython* (2015, O'Reilly).

---

# Математические инструменты

Математики — это пастыри современного мира.

*Билл Гедде*

С появлением профессиональных математиков на Уолл-стрит в 1980-х годах финансы превратились в дисциплину прикладной математики. Если ранние исследовательские работы в области финансов содержали пространственные текстовые изложения и всего несколько математических формул, то современные труды финансовых аналитиков в основном состоят из математических выкладок с редкими вкраплениями текстовых комментариев.

В этой главе мы познакомимся с математическим аппаратом финансовых расчетов, но не будем углубляться в чрезмерные детали. Для этого есть специализированная литература, мы же сконцентрируемся на инструментах Python.

## *Аппроксимация*

В финансовых расчетах чаще всего применяются такие численные методы, как регрессия и интерполяция.

## *Выпуклое программирование*

Задачи выпуклого программирования встречаются во многих финансовых дисциплинах (например, в анализе деривативов при калибровке модели).

## *Интегрирование*

Оценка финансовых активов (деривативов) часто сводится к вычислению интегралов.

## *Символьные вычисления*

SymPy — это мощный пакет символьных вычислений, применяемый, в частности, для решения (систем) уравнений.



# Аппроксимация

Начнем со стандартных операций импорта.

```
In [1]: import numpy as np
        from pylab import plt, mpl
```

```
In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

В этом разделе мы будем использовать следующую функцию, возвращающую результат математического выражения.

```
In [3]: def f(x):
        return np.sin(x) + 0.5 * x
```

Наша задача — аппроксимировать эту функцию в заданном интервале *регрессионными и интерполяционными* методами. Но сначала необходимо построить график функции, чтобы получить представление о том, каким должен быть результат аппроксимации. Мы будем работать в интервале  $[-2\pi, 2\pi]$ . На рис. 11.1 показано, как выглядит график функции в интервале, который задается с помощью функции `np.linspace()`. Мы также создадим вспомогательную функцию `create_plot()`, предназначенную для построения аналогичных графиков в последующих примерах.

```
In [4]: def create_plot(x, y, styles, labels, axlabels):
        plt.figure(figsize=(10, 6))
        for i in range(len(x)):
            plt.plot(x[i], y[i], styles[i], label=labels[i])
            plt.xlabel(axlabels[0])
            plt.ylabel(axlabels[1])
        plt.legend(loc=0)
```

```
In [5]: x = np.linspace(-2 * np.pi, 2 * np.pi, 50) ❶
```

```
In [6]: create_plot([x], [f(x)], ['b'], ['f(x)'], ['x', 'f(x)'])
```

❶ Значения  $x$  для построения графика и вычислений.

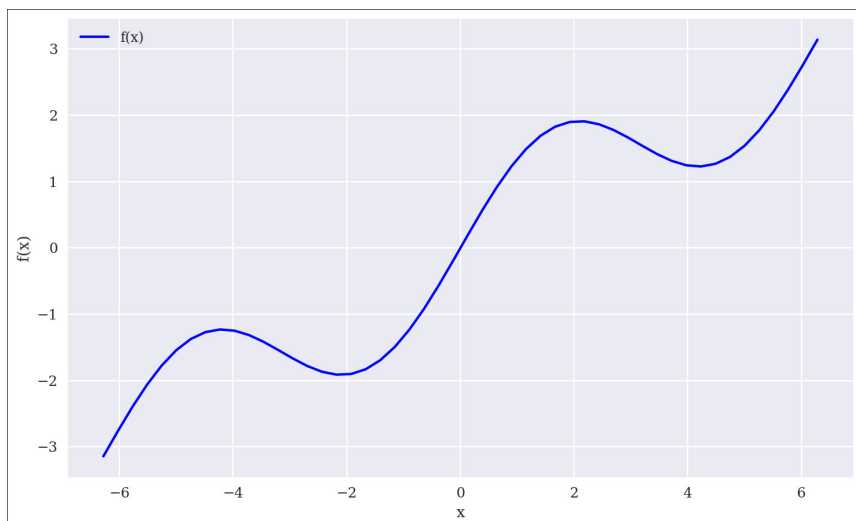


Рис. 11.1. График тестовой функции

## Регрессия

Регрессия — эффективный метод аппроксимации, который хорошо подходит не только для линейных функций, но и для функций более высокого порядка. Численные методы, лежащие в основе регрессии, легко реализуются в коде и обеспечивают высокую скорость вычислений. В общем случае регрессионная задача для набора так называемых базисных функций  $b_d$ ,  $d \in \{1, \dots, D\}$ , заключается в поиске оптимальных параметров  $\alpha_1^*$ , ...,  $\alpha_D^*$ , уравнения 11.1, где  $y_i \equiv f(x_i)$  для  $i \in \{1, \dots, I\}$  наблюдений. Значения  $x_i$  считаются *независимыми наблюдениями*, а  $y_i$  — *зависимыми* (с функциональной или статистической точки зрения).

**Уравнение 11.1.** Регрессионная задача минимизации

$$\min_{\alpha_1, \dots, \alpha_D} \frac{1}{I} \sum_{i=1}^I \left( y_i - \sum_{d=1}^D \alpha_d b_d(x_i) \right)^2.$$

### Базисные функции, выраженные одночленами

Простейший случай — представление базисных функций одночленами:  $b_1 = 1$ ,  $b_2 = x$ ,  $b_3 = x^2$ ,  $b_4 = x^3$  и т.д. В NumPy имеются соответствующие функции, предназначенные как для нахождения оптимальных параметров

(`np.polyfit()`), так и для аппроксимации заданного набора исходных значений (`np.polyval()`).

Параметры функции `np.polyfit()` описаны в табл. 11.1. Подставив в функцию `np.polyval(p, x)` оптимальные регрессионные коэффициенты `p`, возвращаемые функцией `np.polyfit()`, можно получить регрессионные значения для координаты `x`.

*Таблица 11.1. Параметры функции `polyfit()`*

Параметр	Описание
<code>x</code>	Координата <code>x</code> (значения независимой переменной)
<code>y</code>	Координата <code>y</code> (значения зависимой переменной)
<code>deg</code>	Степень регрессионного уравнения
<code>full</code>	Если равен <code>True</code> , дополнительно возвращаются диагностические сведения
<code>w</code>	Веса, применяемые к координатам <code>y</code>
<code>cov</code>	Если равен <code>True</code> , дополнительно возвращается ковариационная матрица

Ниже показан типичный векторизованный способ записи линейного регрессионного уравнения (`deg=1`) с помощью функций `np.polyfit()` и `np.polyval()`. Получив массив регрессионных значений `ry`, мы можем сравнить результаты регрессии с исходной функцией (рис. 11.2). Понятно, что линейная регрессия не способна аппроксимировать тригонометрический член функции.

```
In [7]: res = np.polyfit(x, f(x), deg=1, full=True) ❶
```

```
In [8]: res ❷
```

```
Out[8]: (array([ 4.28841952e-01, -1.31499950e-16]),
         array([21.03238686]),
         2,
         array([1., 1.]),
         1.1102230246251565e-14)
```

```
In [9]: ry = np.polyval(res[0], x) ❸
```

```
In [10]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                    ['f(x)', 'Регрессия'], ['x', 'f(x)'])
```

- ❶ Линейная регрессия.
- ❷ Получение полной информации: параметров регрессии, остатков, ранга матрицы, сингулярных значений и относительного числа обусловленности.
- ❸ Получение регрессионных значений с использованием регрессионных параметров.

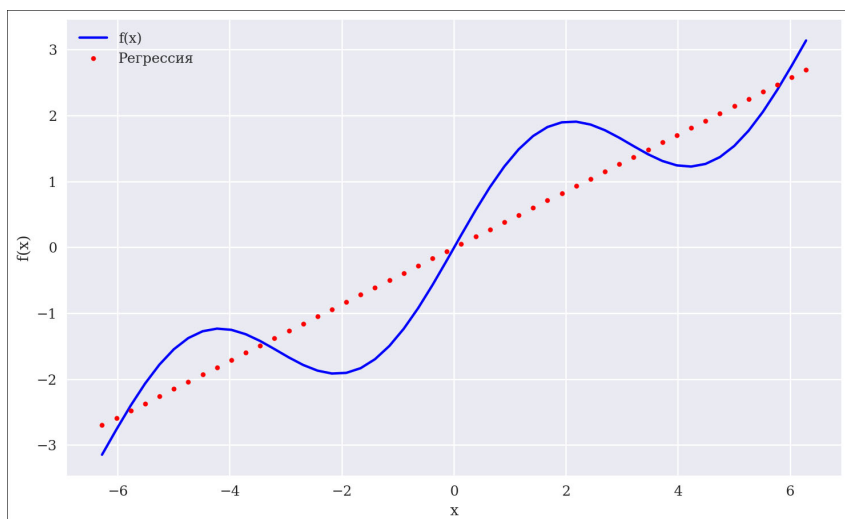


Рис. 11.2. Линейная регрессия

Чтобы аппроксимировать функцию  $\sin()$ , необходимо обратиться к регрессионным уравнениям более высокого порядка. В следующем примере используются базисные функции пятого порядка. Не удивительно, что регрессионная кривая теперь намного точнее повторяет исходную функцию (рис. 11.3). Тем не менее расхождения все еще достаточно значительны.

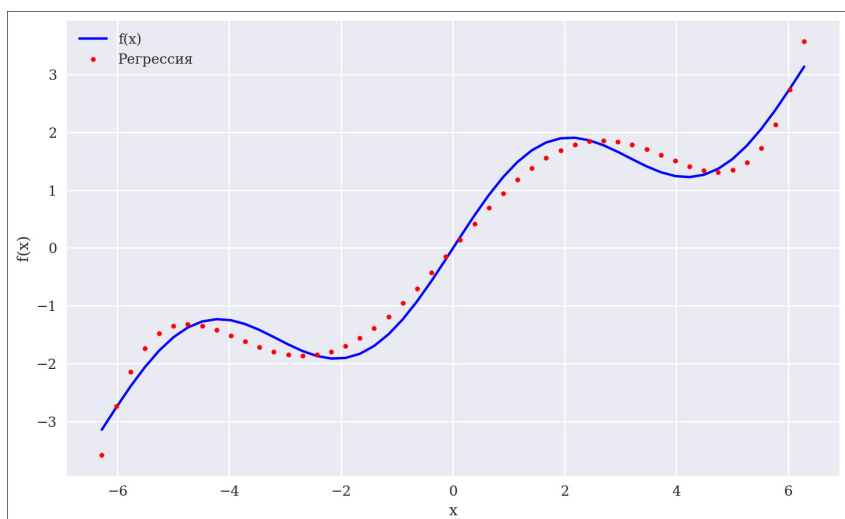


Рис. 11.3. Регрессия с одночленами пятого порядка

```
In [11]: reg = np.polyfit(x, f(x), deg=5)
        ry = np.polyval(reg, x)
```

```
In [12]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                    ['f(x)', 'Регрессия'], ['x', 'f(x)'])
```

В последнем варианте регрессии применяются одночлены 7-го порядка. Результаты говорят сами за себя (рис. 11.4).

```
In [13]: reg = np.polyfit(x, f(x), 7)
        ry = np.polyval(reg, x)
```

```
In [14]: np.allclose(f(x), ry) ❶
Out[14]: False
```

```
In [15]: np.mean((f(x) - ry) ** 2) ❷
Out[15]: 0.0017769134759517689
```

```
In [16]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                    ['f(x)', 'Регрессия'], ['x', 'f(x)'])
```

- ❶ Проверка, являются ли значения функции и регрессионного уравнения одинаковыми (или, по крайней мере, близкими).
- ❷ Вычисление *среднеквадратической ошибки* регрессионных значений для соответствующих значений исходной функции.

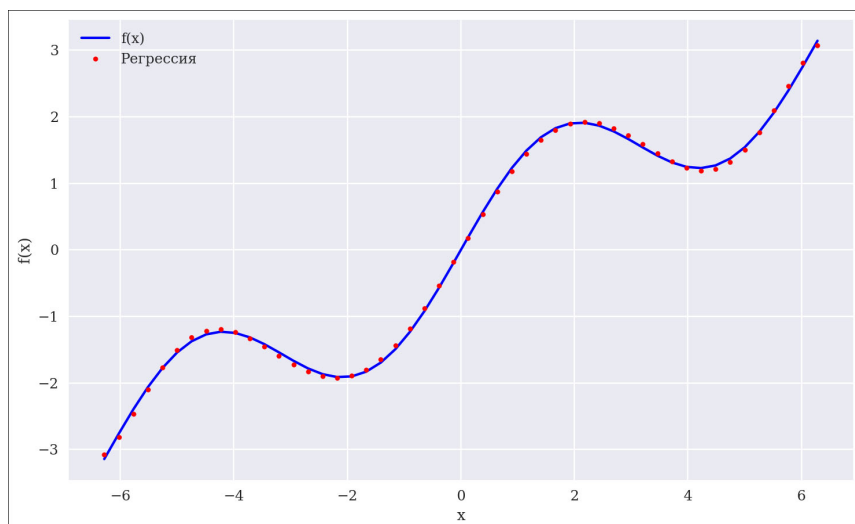


Рис. 11.4. Регрессия с одночленами седьмого порядка

## Отдельные базисные функции

Для получения более точных результатов регрессии следует грамотнее подходить к выбору базисных функций, используя знание аппроксимируемой функции. В нашем случае отдельные базисные функции следует задавать в виде матрицы (т.е. с помощью объекта `ndarray` библиотеки NumPy). Вот как будет выглядеть такая матрица для уравнения с одночленами третьего порядка (рис. 11.5). Регрессионное уравнение здесь вычисляется функцией `np.linalg.lstsq()`.

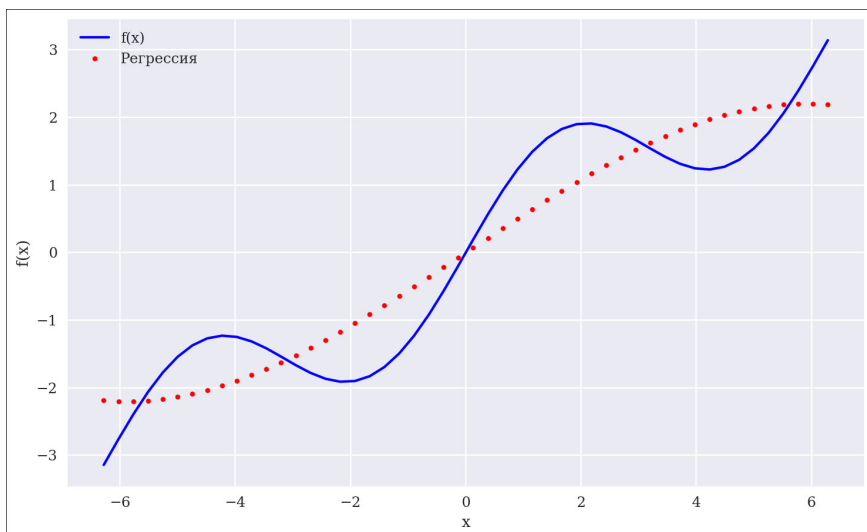


Рис. 11.5. Регрессия с отдельными базисными функциями

```
In [17]: matrix = np.zeros((3 + 1, len(x))) ❶  
         matrix[3, :] = x ** 3 ❷  
         matrix[2, :] = x ** 2 ❷  
         matrix[1, :] = x ❷  
         matrix[0, :] = 1 ❷
```

```
In [18]: reg = np.linalg.lstsq(matrix.T, f(x), rcond=None)[0] ❸
```

```
In [19]: reg.round(4) ❹  
Out[19]: array([ 0.      ,  0.5628, -0.      , -0.0054])
```

```
In [20]: ry = np.dot(reg, matrix) ❺
```

```
In [21]: create_plot([x, x], [f(x), ry], ['b', 'r.'],  
                    ['f(x)', 'Регрессия'], ['x', 'f(x)'])
```

- ❶ Объект `ndarray`, представляющий матрицу значений для базисных функций.
- ❷ Базисные функции от свободного члена до члена третьего порядка.
- ❸ Расчет регрессионного уравнения.
- ❹ Оптимальные параметры регрессии.
- ❺ Регрессионная оценка значений функции.

Показанный на рис. 11.5 результат далек от ожидаемого. Традиционный подход предполагает использование знаний об исходной функции. В частности, нам известно, что она содержит член `sin()`. Поэтому имеет смысл включить соответствующий элемент в набор базисных функций. Для простоты заменим одночлен самого высокого порядка. В результате достигается идеальное совпадение (рис. 11.6).

```
In [22]: matrix[3, :] = np.sin(x) ❶
```

```
In [23]: reg = np.linalg.lstsq(matrix.T, f(x), rcond=None)[0]
```

```
In [24]: reg.round(4) ❷
```

```
Out[24]: array([0. , 0.5, 0. , 1. ])
```

```
In [25]: ry = np.dot(reg, matrix)
```

```
In [26]: np.allclose(f(x), ry) ❸
```

```
Out[26]: True
```

```
In [27]: np.mean((f(x) - ry) ** 2) ❸
```

```
Out[27]: 3.404735992885531e-31
```

```
In [28]: create_plot([x, x], [f(x), ry], ['b', 'r.'],  
                    ['f(x)', 'Регрессия'], ['x', 'f(x)'])
```

- ❶ Новая базисная функция, описывающая известное поведение исходной функции.
- ❷ Оптимальные параметры регрессии.
- ❸ Точность регрессии оказывается идеальной.

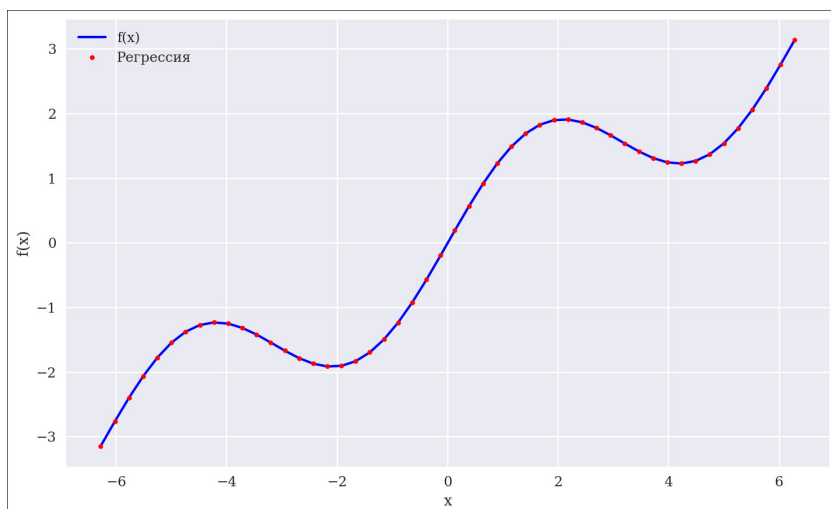


Рис. 11.6. Регрессия с базисной функцией синуса

## Зашумленные данные

Регрессионные методы прекрасно работают даже с зашумленными данными, полученными в результате некорректного моделирования или (неточных) измерений. В качестве иллюстрации сгенерируем независимые и зависимые наблюдения, включающие шум. Как видно на рис. 11.7, результаты регрессии оказались близки к исходной функции. В определенном смысле регрессия усредняет шум до какого-то уровня.

```
In [29]: xn = np.linspace(-2 * np.pi, 2 * np.pi, 50) ❶
         xn = xn + 0.15 * np.random.standard_normal(len(xn)) ❷
         yn = f(xn) + 0.25 * np.random.standard_normal(len(xn)) ❸
```

```
In [30]: reg = np.polyfit(xn, yn, 7)
         ry = np.polyval(reg, xn)
```

```
In [31]: create_plot([x, xn], [f(x), ry], ['b', 'r.',
         ['f(x)', 'Регрессия'], ['x', 'f(x)']])
```

- ❶ Новые детерминированные значения  $x$ .
- ❷ Добавление шума к значениям  $x$ .
- ❸ Добавление шума к значениям  $y$ .



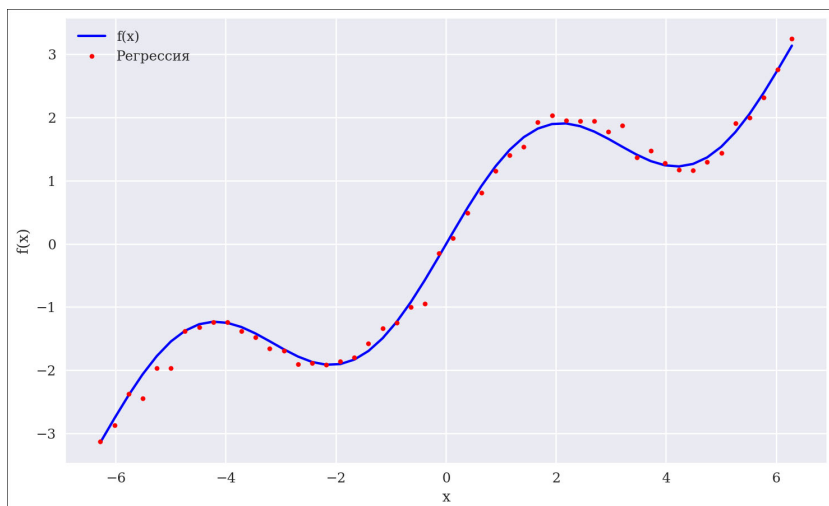


Рис. 11.7. Регрессия с зашумленными данными

## Неотсортированные данные

Еще один важный аспект регрессии состоит в том, что она позволяет работать с неотсортированными данными. В предыдущих примерах мы имели дело с отсортированными значениями  $x$ , что вовсе не обязательно. Попробуем убедиться в этом, рандомизировав значения  $x$ . В следующем примере исходные данные никак не упорядочены.

```
In [32]: xu = np.random.rand(50) * 4 * np.pi - 2 * np.pi ❶
        yu = f(xu)

In [33]: print(xu[:10].round(2))
        print(yu[:10].round(2))
        [-4.17 -0.11 -1.91  2.33  3.34 -0.96  5.81  4.92 -4.56 -5.42]
        [-1.23 -0.17 -1.9  1.89  1.47 -1.29  2.45  1.48 -1.29 -1.95]

In [34]: reg = np.polyfit(xu, yu, 5)
        ry = np.polyval(reg, xu)

In [35]: create_plot([xu, xu], [yu, ry], ['b.', 'ro'],
                    ['f(x)', 'Регрессия'], ['x', 'f(x)'])
```

❶ Рандомизация значений  $x$ .

Как и в случае с зашумленными данными, регрессионный подход не учитывает порядок наблюдений. Это изначально следует из описания задачи минимизации (см. уравнение 11.1) и также иллюстрируется результатами, представленными на рис. 11.8.

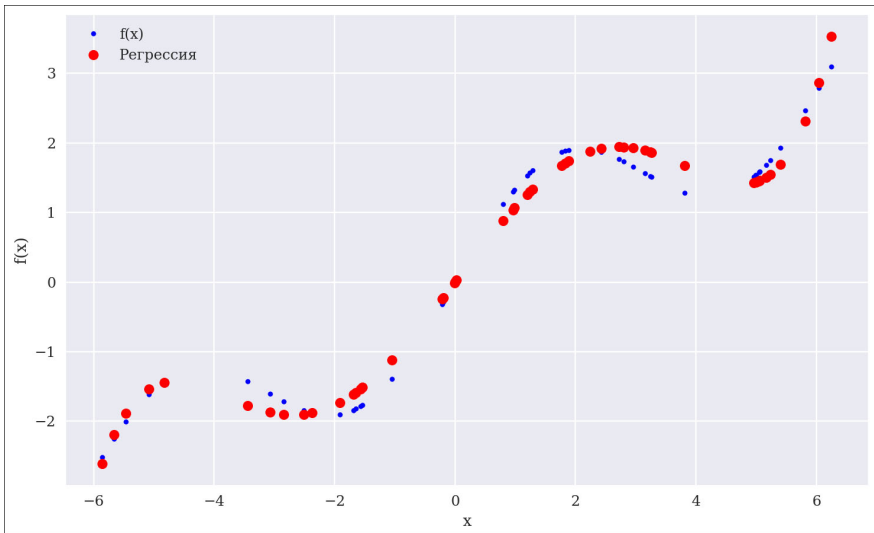


Рис. 11.8. Регрессия неотсортированных данных

## Регрессия многомерных зависимостей

Еще одно удобство регрессионного подхода по методу наименьших квадратов заключается в том, что он легко переносится на многомерные зависимости, не требуя существенных модификаций. В качестве примера рассмотрим следующую функцию  $f_m()$ .

```
In [36]: def fm(p):
          x, y = p
          return np.sin(x) + 0.25 * x + np.sqrt(y) + 0.05 * y ** 2
```

Чтобы корректно визуализировать эту функцию, мы должны оперировать двухмерными сетками независимых наблюдений. График функции  $f_m()$  в трехмерном координатном пространстве XYZ показан на рис. 11.9.

```
In [37]: x = np.linspace(0, 10, 20)
          y = np.linspace(0, 10, 20)
          X, Y = np.meshgrid(x, y) ❶
```

```

In [38]: Z = fm((X, Y))
         x = X.flatten() ❷
         y = Y.flatten() ❷

In [39]: from mpl_toolkits.mplot3d import Axes3D ❸

In [40]: fig = plt.figure(figsize=(12, 8))
         ax = fig.gca(projection='3d')
         surf = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                                cmap='coolwarm', linewidth=0.5,
                                antialiased=True)

         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('f(x, y)')
         fig.colorbar(surf, shrink=0.5, aspect=5)

```

- ❶ Генерирование двухмерных объектов `ndarray` (“сеток”) из одномерных объектов `ndarray`.
- ❷ Получение одномерных объектов `ndarray` из двухмерных объектов `ndarray`.
- ❸ Импорт инструментов 3D-визуализации из библиотеки `matplotlib`.

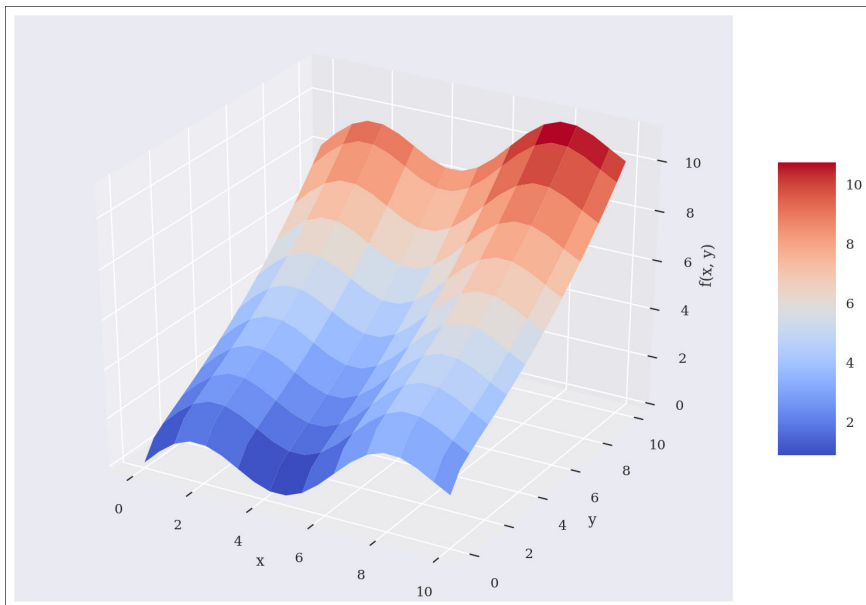


Рис. 11.9. График функции двух переменных

Чтобы получить хорошие результаты регрессии, нужно правильно подобрать базисные функции. Зная структуру функции `fm()`, мы выбираем в качестве базисных функции `np.sin()` и `np.sqrt()`. Результаты регрессии представлены на рис. 11.10.

```
In [41]: matrix = np.zeros((len(x), 6 + 1))
         matrix[:, 6] = np.sqrt(y) ❶
         matrix[:, 5] = np.sin(x) ❷
         matrix[:, 4] = y ** 2
         matrix[:, 3] = x ** 2
         matrix[:, 2] = y
         matrix[:, 1] = x
         matrix[:, 0] = 1

In [42]: reg = np.linalg.lstsq(matrix, fm((x, y)), rcond=None)[0]

In [43]: RZ = np.dot(matrix, reg).reshape((20, 20)) ❸

In [44]: fig = plt.figure(figsize=(12, 8))
         ax = fig.gca(projection='3d')
         surf1 = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                                cmap=mpl.cm.coolwarm, linewidth=0.5,
                                antialiased=True) ❹
         surf2 = ax.plot_wireframe(X, Y, RZ, rstride=2,
                                   cstride=2, label='Регрессия') ❺

         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('f(x, y)')
         ax.legend()
         fig.colorbar(surf, shrink=0.5, aspect=5)
```

- ❶ Функция `np.sqrt()` для переменной `y`.
- ❷ Функция `np.sin()` для переменной `x`.
- ❸ Преобразование результатов регрессии в двумерную структуру (сетку).
- ❹ Построение поверхности, представляющей исходную функцию.
- ❺ Построение регрессионной поверхности.

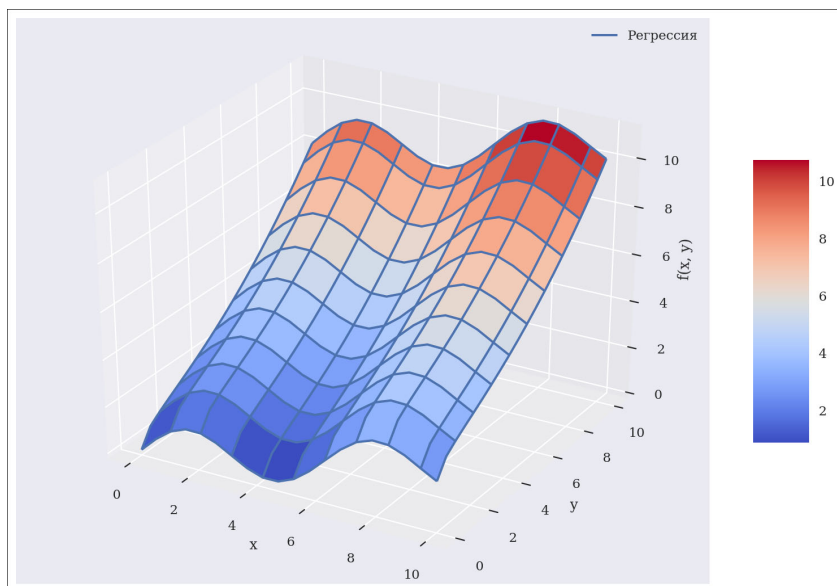


Рис. 11.10. Регрессионная поверхность для функции двух переменных



## Регрессия

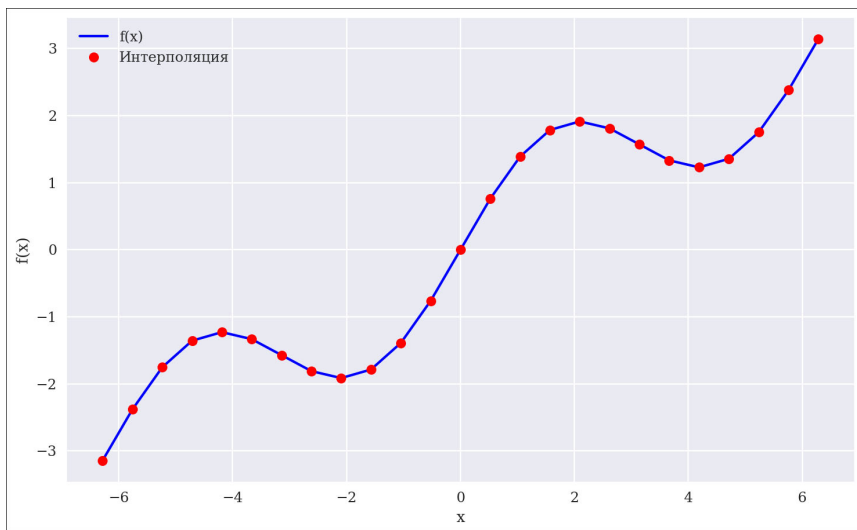
Регрессия по методу наименьших квадратов имеет несколько областей применения, включая аппроксимацию простых функций, а также аппроксимацию функций на основе зашумленных или неотсортированных данных. Такой подход можно применять как к одномерным, так и к многомерным задачам. Математический аппарат при этом остается практически неизменным.

## Интерполяция

По сравнению с регрессией *интерполяция* (например, кубическими сплайнами) требует более сложных математических расчетов. Она также ограничена лишь задачами с малым числом размерностей. Для набора упорядоченных (по координате  $X$ ) наблюдений интерполяция предполагает такую регрессию между двумя соседними точками, чтобы точки данных не только идеально соответствовали результирующей кусочно-определенной интерполяционной функции, но и обеспечивали непрерывную дифференцируемость такой функции в рассматриваемом диапазоне значений. Чтобы получить непрерывно дифференцируемую функцию, интерполяцию нужно

выполнять с помощью многочленов не менее чем третьего порядка, т.е. с помощью кубических сплайнов. Тем не менее в большинстве случаев хороший результат можно получить при интерполяции квадратичными и даже линейными сплайнами.

В следующем примере выполняется линейная интерполяция (рис. 11.11).



*Рис. 11.11. Интерполяция линейными сплайнами (полный набор значений)*

```
In [45]: import scipy.interpolate as spi ❶

In [46]: x = np.linspace(-2 * np.pi, 2 * np.pi, 25)

In [47]: def f(x):
           return np.sin(x) + 0.5 * x

In [48]: ipo = spi.splrep(x, f(x), k=1) ❷

In [49]: iy = spi.splev(x, ipo) ❸

In [50]: np.allclose(f(x), iy) ❹
Out[50]: True

In [51]: create_plot([x, x], [f(x), iy], ['b', 'ro'],
                     ['f(x)', 'Интерполяция'], ['x', 'f(x)'])
```

- ❶ Импорт необходимого модуля из пакета SciPy.
- ❷ Интерполяция линейными сплайнами.
- ❸ Получение интерполированных значений.
- ❹ Проверка близости интерполированных значений к соответствующим значениям функции.

Учитывая упорядоченность значений  $x$ , практическая реализация рассмотренного выше метода оказалась столь же простой, как и в случае использования функций `np.polyfit()` и `np.polyval()`. Все необходимые вычисления выполняются функциями `sci.splrep()` и `sci.splev()`. В табл. 11.2 приведено описание основных параметров функции `sci.splrep()`.

Таблица 11.2. Параметры функции `sci.splrep()`

Параметр	Описание
<code>x</code>	Упорядоченные значения координаты $x$ (независимая переменная)
<code>y</code>	Значения координаты $y$ (зависимая переменная) для упорядоченных значений $x$
<code>w</code>	Веса для координат $y$
<code>xb, xe</code>	Шаг интерполяции; в случае <code>None</code> принимается равным <code>[x[0], x[-1]]</code>
<code>k</code>	Порядок сплайна ( $1 \leq k \leq 5$ )
<code>S</code>	Коэффициент сглаживания (чем он больше, тем более плавной получается функция)
<code>full_output</code>	Если равен <code>True</code> , выводятся дополнительные сведения о выполненной интерполяции
<code>quiet</code>	Значение <code>True</code> предотвращает вывод диагностических сообщений

Параметры функции `sci.splev()` описаны в табл. 11.3.

Таблица 11.3. Параметры функции `sci.splev()`

Параметр	Описание
<code>x</code>	Упорядоченные значения координаты $x$ (независимая переменная)
<code>tck</code>	Последовательность из трех элементов, возвращаемая функцией <code>sci.splrep()</code> (узлы, коэффициенты, степень)
<code>der</code>	Порядок производной (0 — функция; 1 — первая производная)
<code>ext</code>	Способ обработки значений $x$ , заданных вне узлов (0 — экстраполяция, 1 — представление значением 0, 2 — ошибка <code>ValueError</code> )

В финансовых расчетах интерполяция сплайнами часто применяется для оценки зависимых значений, получаемых для независимых точек данных, не представленных в наборе исходных наблюдений. В качестве иллюстрации рассмотрим следующий пример, в котором интерполяция линейными сплайнами осуществляется с очень коротким шагом. На рис. 11.12 видно, что значения между двумя соседними наблюдениями подбираются *линейным* образом. Далеко не во всех приложениях такая точность будет достаточной. Другим недостатком становится то, что результирующая функция не является непрерывно дифференцируемой в точках исходных наблюдений.

```
In [52]: xd = np.linspace(1.0, 3.0, 50) ❶  
         iyd = spi.splev(xd, ipo)
```

```
In [53]: create_plot([xd, xd], [f(xd), iyd], ['b', 'ro'],  
                    ['f(x)', 'Интерполяция'], ['x', 'f(x)'])
```

❶ Меньший шаг, большее количество точек интерполяции.

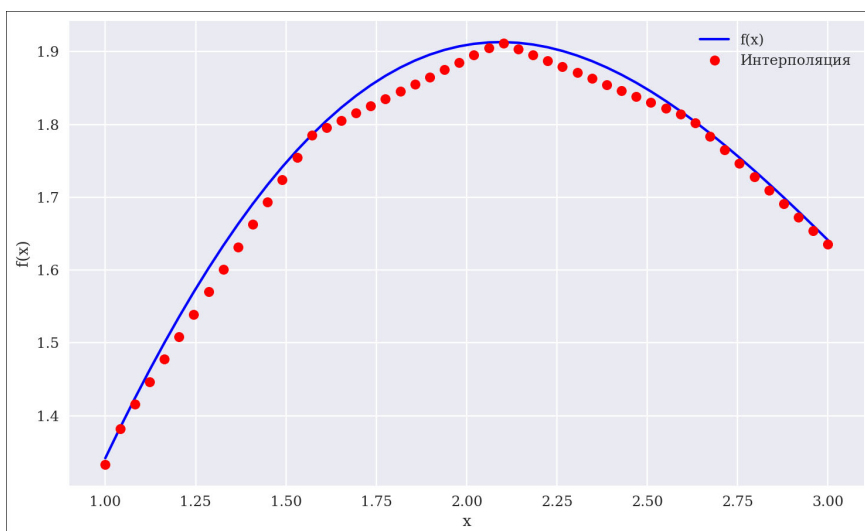


Рис. 11.12. Интерполяция линейными сплайнами (подмножество данных)

Повторная интерполяция этого же набора данных, но уже кубическими сплайнами, позволяет получить более точные результаты (рис. 11.13).

```
In [54]: ipo = spi.splrep(x, f(x), k=3) ❶  
         iyd = spi.splev(xd, ipo) ❷
```



```
In [55]: np.allclose(f(xd), iyd) ❸  
Out[55]: False
```

```
In [56]: np.mean((f(xd) - iyd) ** 2) ❹  
Out[56]: 1.1349319851436892e-08
```

```
In [57]: create_plot([xd, xd], [f(xd), iyd], ['b', 'ro'],  
                    ['f(x)', 'Интерполяция'], ['x', 'f(x)'])
```

- ❶ Интерполяция кубическими сплайнами всего набора данных.
- ❷ Интерполяция для меньшего интервала.
- ❸ Интерполяция по-прежнему не идеальна...
- ❹ ...но намного точнее, чем в предыдущих случаях.

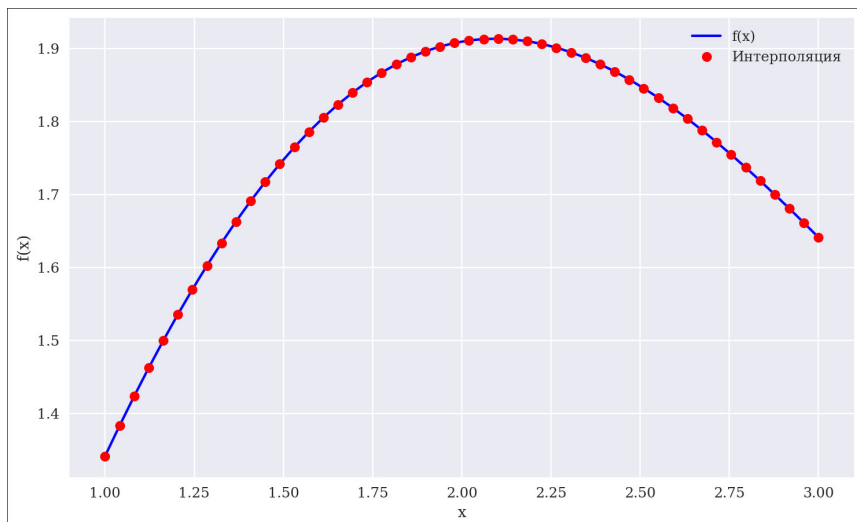


Рис. 11.13. Интерполяция кубическими сплайнами  
(подмножество данных)



### Интерполяция

Интерполяция сплайнами позволяет получить более точные результаты, чем регрессия по методу наименьших квадратов. Но не забывайте, что для такой интерполяции требуются отсортированные (и незашумленные) данные, к тому же она применима только

к задачам с малым числом размерностей. С вычислительной точки зрения это более трудоемкий подход, во многих случаях работающий медленнее, чем регрессия.

## Выпуклое программирование

*Выпуклое программирование* играет важную роль в финансовой математике. В качестве примера можно привести калибровку моделей оценки опционов на основе рыночных данных или оптимизацию функции полезности. В этом разделе мы будем работать со следующей функцией.

```
In [58]: def fm(p):  
         x, y = p  
         return (np.sin(x) + 0.05 * x ** 2 + np.sin(y) + \  
                 0.05 * y ** 2)
```

График функции в заданных интервалах значений  $x$  и  $y$  показан на рис. 11.14. Визуальная оценка графика позволяет сделать вывод о том, что функция характеризуется большим количеством локальных минимумов. Не ясно, существует ли у функции глобальный минимум, но очень похоже на то.

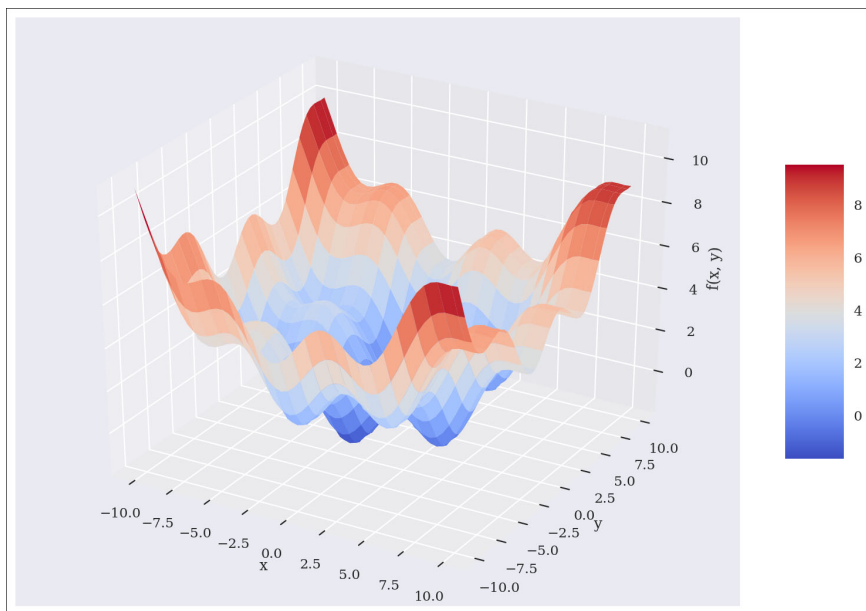


Рис. 11.14. График оптимизируемой функции

```
In [59]: x = np.linspace(-10, 10, 50)
        y = np.linspace(-10, 10, 50)
        X, Y = np.meshgrid(x, y)
        Z = fm((X, Y))

In [60]: fig = plt.figure(figsize=(12, 8))
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                               cmap='coolwarm', linewidth=0.5,
                               antialiased=True)

        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('f(x, y)')
        fig.colorbar(surf, shrink=0.5, aspect=5)
```

## Глобальная оптимизация

В следующих примерах мы попытаемся решить задачи глобальной и локальной оптимизации. Используемые нами функции `sco.brute()` и `sco.fmin()` содержатся в модуле `scipy.optimize`.

Чтобы наглядно продемонстрировать, как именно выполняется процедура минимизации, мы добавим в нашу функцию параметр, позволяющий вывести текущие значения независимых переменных и самой функции. Это даст возможность отслеживать изменения на каждом шаге.

```
In [61]: import scipy.optimize as sco ❶

In [62]: def fo(p):
        x, y = p
        z = np.sin(x) + 0.05 * x ** 2 + np.sin(y) + \
            0.05 * y ** 2
        if output == True:
            print('%8.4f | %8.4f | %8.4f' % (x, y, z)) ❷
        return z

In [63]: output = True
        sco.brute(fo, ((-10, 10.1, 5), (-10, 10.1, 5)), finish=None) ❸

-10.0000 | -10.0000 | 11.0880
-10.0000 | -10.0000 | 11.0880
-10.0000 | -5.0000 | 7.7529
-10.0000 | 0.0000 | 5.5440
-10.0000 | 5.0000 | 5.8351
-10.0000 | 10.0000 | 10.0000
```

-5.0000		-10.0000		7.7529
-5.0000		-5.0000		4.4178
-5.0000		0.0000		2.2089
-5.0000		5.0000		2.5000
-5.0000		10.0000		6.6649
0.0000		-10.0000		5.5440
0.0000		-5.0000		2.2089
0.0000		0.0000		0.0000
0.0000		5.0000		0.2911
0.0000		10.0000		4.4560
5.0000		-10.0000		5.8351
5.0000		-5.0000		2.5000
5.0000		0.0000		0.2911
5.0000		5.0000		0.5822
5.0000		10.0000		4.7471
10.0000		-10.0000		10.0000
10.0000		-5.0000		6.6649
10.0000		0.0000		4.4560
10.0000		5.0000		4.7471
10.0000		10.0000		8.9120

Out[63]: array([0., 0.])

- ❶ Импорт необходимого модуля из пакета SciPy.
- ❷ Сведения, выводимые, если параметр `output` равен `True`.
- ❸ Оптимизация методом грубой силы.

Оптимальные значения переменных, учитывая начальную параметризацию функции, определяются как  $x = y = 0$ . Значение функции в данной точке тоже равно 0, как видно из таблицы результатов. Существует большой соблазн объявить это значение глобальным минимумом. Но не стоит забывать, что параметризация выполнена довольно грубо: шаг изменения обоих входных параметров равен 5. Для получения более точных результатов следует уменьшить шаг. В итоге оказывается, что предыдущее решение не было оптимальным.

```
In [64]: output = False
         opt1 = sco.brute(fo, ((-10, 10.1, 0.1), (-10, 10.1, 0.1)),
                        finish=None)
```

```
In [65]: opt1
```

```
Out[65]: array([-1.4, -1.4])
```

```
In [66]: fm(opt1)
Out[66]: -1.7748994599769203
```

Как видите, теперь оптимальные значения переменных равны  $x = y = -1.4$ , а глобальный минимум функции примерно равен  $-1.7749$ .

## Локальная оптимизация

При выполнении локальной оптимизации мы используем результаты глобальной оптимизации. Функция `sco.fmin()` получает в качестве аргументов минимизируемую функцию и начальные значения параметров. Необязательными аргументами являются допуски по входным параметрам и значению функции, а также максимальное количество итераций и вызовов функции. Локальная оптимизация позволяет дополнительно уточнить результаты.

```
In [67]: output = True
         opt2 = sco.fmin(fo, opt1, xtol=0.001, ftol=0.001,
                        maxiter=15, maxfun=20) ❶
```

-1.4000		-1.4000		-1.7749
-1.4700		-1.4000		-1.7743
-1.4000		-1.4700		-1.7743
-1.3300		-1.4700		-1.7696
-1.4350		-1.4175		-1.7756
-1.4350		-1.3475		-1.7722
-1.4088		-1.4394		-1.7755
-1.4438		-1.4569		-1.7751
-1.4328		-1.4427		-1.7756
-1.4591		-1.4208		-1.7752
-1.4213		-1.4347		-1.7757
-1.4235		-1.4096		-1.7755
-1.4305		-1.4344		-1.7757
-1.4168		-1.4516		-1.7753
-1.4305		-1.4260		-1.7757
-1.4396		-1.4257		-1.7756
-1.4259		-1.4325		-1.7757
-1.4259		-1.4241		-1.7757
-1.4304		-1.4177		-1.7757
-1.4270		-1.4288		-1.7757

```
Warning: Maximum number of function evaluations has been
exceeded.
```

```
In [68]: opt2
Out[68]: array([-1.42702972, -1.42876755])
```

```
In [69]: fm(opt2)
Out[69]: -1.7757246992239009
```

### ❶ Локальная оптимизация.

Во многих задачах выпуклого программирования лучше сначала выполнять глобальную минимизацию и лишь затем переходить к локальной. Дело в том, что алгоритмы локальной минимизации могут попасть в ловушку локального экстремума, проигнорировав другие возможные экстремумы и/или глобальный минимум. В следующем примере показано, что при начальной параметризации  $x = y = 2$  за “минимум” функции ошибочно принимается значение выше нуля.

```
In [70]: output = False
         sco.fmin(f0, (2.0, 2.0), maxiter=250)
         Optimization terminated successfully.
             Current function value: 0.015826
             Iterations: 46
             Function evaluations: 86

Out[70]: array([4.2710728 , 4.27106945])
```

## Условная оптимизация

Ранее мы имели дело только с задачами безусловной оптимизации. Однако огромное количество экономических и финансовых задач оптимизации ограничивается одним или несколькими условиями. В математике такие ограничения обычно представляются равенствами или неравенствами.

В качестве примера рассмотрим задачу максимизации полезности (поиска максимальной ожидаемой полезности) инвестиций в два портфеля ценных бумаг. Стоимость каждого портфеля —  $q_a = q_b = 10$  долларов. Спустя год по ним выплачивается доход: 15 и 5 долларов при исходе  $u$ , а также 5 и 12 долларов при исходе  $d$ . Оба исхода равновероятны. Обозначим векторы выплат соответственно  $r_a$  и  $r_b$ .

Инвестор располагает бюджетом  $w_0 = 100$  долларов, а доход рассчитывается по формуле полезности  $u(w) = \sqrt{w}$ , где  $w$  — вкладываемые средства (в долларах). Математическая постановка задачи максимизации, где  $a$  и  $b$  — количество ценных бумаг, приобретаемых инвестором в каждом портфеле, приведена в уравнении 11.2.

### Уравнение 11.2. Задача максимизации ожидаемой полезности (1)

$$\begin{aligned}\max_{a,b} \mathbf{E}(u(w_1)) &= p\sqrt{w_{1u}} + (1-p)\sqrt{w_{1d}}, \\ w_1 &= ar_a + br_b, \\ w_0 &\geq aq_a + bq_b, \\ a, b &\geq 0.\end{aligned}$$

Подставив в эту систему все известные числовые значения, ее можно привести к следующему виду (обратите внимание на переход к задаче минимизации отрицательной ожидаемой полезности).

### Уравнение 11.3. Задача максимизации ожидаемой полезности (2)

$$\begin{aligned}\min_{a,b} -\mathbf{E}(u(w_1)) &= -(0,5\sqrt{w_{1u}} + 0,5\sqrt{w_{1d}}), \\ w_{1u} &= a \cdot 15 + b \cdot 5, \\ w_{1d} &= a \cdot 5 + b \cdot 12, \\ 100 &\geq a \cdot 10 + b \cdot 10, \\ a, b &\geq 0.\end{aligned}$$

В решении задачи нам поможет функция `scipy.optimize.minimize()`. В качестве аргументов она получает не только оптимизируемую функцию, но и условия в виде равенств и неравенств, ограничивающих оптимизацию (список словарей), а также граничные значения параметров (кортеж кортежей)<sup>1</sup>. Ниже показано, как перевести уравнение 11.3 в код Python.

```
In [71]: import math
```

```
In [72]: def Eu(p): ❶
           s, b = p
           return -(0.5 * math.sqrt(s * 15 + b * 5) +
                   0.5 * math.sqrt(s * 5 + b * 12))
```

```
In [73]: cons = ({'type': 'ineq',
                  'fun': lambda p: 100 - p[0] * 10 - p[1] * 10}) ❷
```

```
In [74]: bnds = ((0, 1000), (0, 1000)) ❸
```

---

<sup>1</sup> Подробное описание функции `minimize()` приведено в документации ([http://bit.ly/using\\_minimize](http://bit.ly/using_minimize)).

```
In [75]: result = sco.minimize(Eu, [5, 5], method='SLSQP',  
                               bounds=bnds, constraints=cons) ④
```

- ① Функция, *минимизируемая* в задаче максимизации ожидаемой полезности.
- ② Ограничение в виде неравенства, заданное с помощью объекта `dict`.
- ③ Граничные значения параметров (выбираемые в широком диапазоне).
- ④ Условная оптимизация.

Возвращаемый объект содержит всю необходимую информацию. Что касается минимального значения функции, то нужно помнить о необходимости замены знака на противоположный.

```
In [76]: result  
Out[76]:      fun: -9.700883611487832  
          jac: array([-0.48508096, -0.48489535])  
          message: 'Optimization terminated successfully.'  
          nfev: 21  
          nit: 5  
          njev: 5  
          status: 0  
          success: True  
          x: array([8.02547122, 1.97452878])
```

```
In [77]: result['x'] ①  
Out[77]: array([8.02547122, 1.97452878])
```

```
In [78]: -result['fun'] ②  
Out[78]: 9.700883611487832
```

```
In [79]: np.dot(result['x'], [10, 10]) ③  
Out[79]: 99.99999999999999
```

- ① Оптимальные значения переменных (т.е. оптимальный портфель).
- ② Отрицательное минимальное значение функции как решение задачи оптимизации.
- ③ Достигнут лимит бюджета; инвестируются все средства.



# Интегрирование

К интегрированию часто прибегают в задачах оценки опционов. Это связано с тем, что в общем случае риск-нейтральную стоимость деривативов можно выразить как дисконтированное *ожидание* доходности в риск-нейтральных или мартингальных условиях. В свою очередь, ожидание вычисляется как сумма (дискретная функция) или интеграл (непрерывная функция). Все необходимые нам функции численного интегрирования содержатся в модуле `scipy.integrate`. Функция, на примере которой мы будем знакомиться с методами численного интегрирования в Python, взята из раздела “Аппроксимация”.

```
In [80]: import scipy.integrate as sci
```

```
In [81]: def f(x):  
         return np.sin(x) + 0.5 * x
```

Интервал интегрирования задается равным  $[0,5; 9,5]$ ; в результате получаем определенный интеграл, описываемый уравнением 11.4.

**Уравнение 11.4. Интеграл тестовой функции**

$$\int_{0,5}^{9,5} f(x)dx = \int_{0,5}^{9,5} \left( \sin x + \frac{x}{2} \right) dx.$$

Ниже объявляются объекты Python, требуемые для вычисления такого интеграла.

```
In [82]: x = np.linspace(0, 10)  
         y = f(x)  
         a = 0.5 ❶  
         b = 9.5 ❷  
         Ix = np.linspace(a, b) ❸  
         Iy = f(Ix) ❹
```

- ❶ Нижний предел интегрирования.
- ❷ Верхний предел интегрирования.
- ❸ Значения интервала интегрирования.
- ❹ Значения подынтегральной функции.

Результат интегрирования представляется областью, выделенной серым цветом под графиком интегрируемой функции (рис. 11.15)<sup>2</sup>.

In [83]: `from matplotlib.patches import Polygon`

```
In [84]: fig, ax = plt.subplots(figsize=(10, 6))
plt.plot(x, y, 'b', linewidth=2)
plt.ylim(bottom=0)
Ix = np.linspace(a, b)
Iy = f(Ix)
verts = [(a, 0)] + list(zip(Ix, Iy)) + [(b, 0)]
poly = Polygon(verts, facecolor='0.7', edgecolor='0.5')
ax.add_patch(poly)
plt.text(0.75 * (a + b), 1.5, r»$ \int_a^b f(x)dx$,
        horizontalalignment='center', fontsize=20)
plt.figtext(0.9, 0.075, '$x$')
plt.figtext(0.075, 0.9, '$f(x)$')
ax.set_xticks((a, b))
ax.set_xticklabels(('a', 'b'))
ax.set_yticks([f(a), f(b)]);
```



*Рис. 11.15. Результат интегрирования как площадь области под графиком функции*

<sup>2</sup> О том, как строить такие графики, см. в главе 7.

## Численное интегрирование

Модуль `scipy.integrate` содержит ряд функций, предназначенных для численного интегрирования различных математических функций в заданных нижнем и верхнем пределах. В качестве примеров можно привести функции `sci.fixed_quad()` (*метод Гаусса*), `sci.quad()` (*адаптивная квадратура*) и `sci.romberg()` (*метод Ромберга*).

```
In [85]: sci.fixed_quad(f, a, b)[0]
Out[85]: 24.366995967084602
```

```
In [86]: sci.quad(f, a, b)[0]
Out[86]: 24.374754718086752
```

```
In [87]: sci.romberg(f, a, b)
Out[87]: 24.374754718086713
```

Есть также ряд функций интегрирования, которым в качестве аргументов можно передавать объекты `list` (значения функции) или `ndarray` (значения интервала интегрирования). Например, функция `sci.trapz()` выполняет интегрирование по *методу трапеций*, а функция `sci.simps()` — по *методу Симпсона*.

```
In [88]: xi = np.linspace(0.5, 9.5, 25)
```

```
In [89]: sci.trapz(f(xi), xi)
Out[89]: 24.352733271544516
```

```
In [90]: sci.simps(f(xi), xi)
Out[90]: 24.37496418455075
```

## Интегрирование методами моделирования

Оценка опционов и деривативов по методу Монте-Карло (рассматривается в главе 12) основана на предположении о возможности интегрирования моделируемой функции. Для этого следует выбрать  $I$  случайных значений  $x$  в интервале интегрирования и вычислить значение подынтегральной функции для каждого из них. Далее значения функции суммируются, и вычисляется среднее по интервалу. Умножив полученное среднее на длину интервала, находим оценочное значение интеграла.

Следующие результаты показывают, что оценка интеграла, вычисляемая по методу Монте-Карло, сходится, хоть и не монотонно, к действительному

значению интеграла при увеличении количества случайных выборок. Оценка получается достаточно близкой даже для относительно небольшого числа случайных выборок.

```
In [91]: for i in range(1, 20):
          np.random.seed(1000)
          x = np.random.random(i * 10) * (b - a) + a ❶
          print(np.mean(f(x)) * (b - a))
24.804762279331463
26.522918898332378
26.265547519223976
26.02770339943824
24.99954181440844
23.881810141621663
23.527912274843253
23.507857658961207
23.67236746066989
23.679410416062886
24.424401707879305
24.239005346819056
24.115396924962802
24.424191987566726
23.924933080533783
24.19484212027875
24.117348378249833
24.100690929662274
23.76905109847816
```

❶ Количество случайных значений  $x$  увеличивается на каждой итерации.

## Символьные вычисления

В предыдущих разделах мы имели дело преимущественно с численными методами. В этом разделе вам предстоит познакомиться с *символьными вычислениями*, которые находят широкое применение в финансовой математике. Инструменты символьных вычислений содержатся в библиотеке SymPy.

### Общие сведения

Библиотека SymPy предоставляет собственные классы объектов. Базовый класс называется `Symbol`.

```
In [92]: import sympy as sy
```

```
In [93]: x = sy.Symbol('x') ❶  
         y = sy.Symbol('y') ❶
```

```
In [94]: type(x)  
Out[94]: sympy.core.symbol.Symbol
```

```
In [95]: sy.sqrt(x) ❷  
Out[95]: sqrt(x)
```

```
In [96]: 3 + sy.sqrt(x) - 4 ** 2 ❸  
Out[96]: sqrt(x) - 13
```

```
In [97]: f = x ** 2 + 3 + 0.5 * x ** 2 + 3 / 2 ❹
```

```
In [98]: sy.simplify(f) ❺  
Out[98]: 1.5*x**2 + 4.5
```

- ❶ Определение символов, используемых в вычислениях.
- ❷ Применение функции к символу.
- ❸ Численное выражение с использованием символа.
- ❹ Символьное представление функции.
- ❺ Упрощенная запись функции.

Такой подход заметно отличается от того, к чему мы привыкли. Несмотря на то что у  $x$  нет числового значения, функция вычисления квадратного корня  $x$  все равно определяется с помощью библиотеки SymPy, поскольку  $x$  — это объект `Symbol`. В результате вызов `sy.sqrt(x)` может быть частью произвольного математического выражения. Заметьте, что SymPy автоматически упрощает математические выражения. С помощью объектов `Symbol` можно определять произвольные функции, которые не следует путать с функциями Python.

В SymPy поддерживаются три способа кодировки математических выражений:

- LaTeX;
- Unicode;
- ASCII.

При работе исключительно в среде Jupyter Notebook (на HTML-странице) предпочтителен формат LaTeX как наиболее визуально привлекательный. В следующем примере используется самый простой вариант — ASCII, чтобы подчеркнуть отсутствие ручного форматирования.

```
In [99]: sy.init_printing(pretty_print=False, use_unicode=False)
```

```
In [100]: print(sy.pretty(f))
          2
        1.5*x + 4.5
```

```
In [101]: print(sy.pretty(sy.sqrt(x) + 0.5))
```

$$\sqrt{x} + 0.5$$

Библиотека SymPy включает большое количество других полезных математических функций, например для представления числа  $p$ . В следующем примере показаны первые и последние 40 знаков после запятой в строковом представлении числа  $p$  (доходит до 400 000-го разряда). Также демонстрируется, как найти в записи числа  $p$  шестизначную дату рождения в формате “ддммгг”.

```
In [102]: %time pi_str = str(sy.N(sy.pi, 400000)) ❶
          CPU times: user 400 ms, sys: 10.9 ms, total: 411 ms
          Wall time: 501 ms
```

```
In [103]: pi_str[:42] ❷
Out[103]: '3.1415926535897932384626433832795028841971'
```

```
In [104]: pi_str[-40:] ❸
Out[104]: '8245672736856312185020980470362464176198'
```

```
In [105]: %time pi_str.find('061072') ❹
          CPU times: user 115 µs, sys: 1e+03 ns, total: 116 µs
          Wall time: 120 µs
```

```
Out[105]: 80847
```

- ❶ Получение строкового представления первых 400 000 разрядов числа  $p$ .
- ❷ Отображение первых 40 цифр после запятой...
- ❸ ...и 40 последних цифр.
- ❹ Поиск шестизначной даты рождения в строке.

## Решение уравнений

Одно из достоинств библиотеки SymPy — возможность решения уравнений, таких, например, как  $x^2 - 1 = 0$ . Предполагается, что ищется корень уравнения, в котором заданное выражение равно 0. Следовательно, уравнения вида  $x^2 - 1 = 3$  необходимо предварительно перестроить. Библиотеке по силам справиться и с решением более сложных уравнений, таких как  $x^3 + 0,5x^2 - 1 = 0$ . Наконец, она способна находить комплексные корни в случае уравнений вида  $x^2 + y^2 = 0$ .

```
In [106]: sy.solve(x ** 2 - 1)
```

```
Out[106]: [-1, 1]
```

```
In [107]: sy.solve(x ** 2 - 1 - 3)
```

```
Out[107]: [-2, 2]
```

```
In [108]: sy.solve(x ** 3 + 0.5 * x ** 2 - 1)
```

```
Out[108]: [0.858094329496553, -0.679047164748276 - 0.839206763026694*I,  
          -0.679047164748276 + 0.839206763026694*I]
```

```
In [109]: sy.solve(x ** 2 + y ** 2)
```

```
Out[109]: [{x: -I*y}, {x: I*y}]
```

## Интегрирование

Еще одно достоинство библиотеки SymPy — поддержка интегрирования и дифференцирования. В следующем примере выполняется *численное* и *символьное* интегрирование уже знакомой нам тестовой функции. Предварительно необходимо задать пределы интегрирования в виде объектов Symbol.

```
In [110]: a, b = sy.symbols('a b') ❶
```

```
In [111]: I = sy.Integral(sy.sin(x) + 0.5 * x, (x, a, b)) ❷
```

```
In [112]: print(sy.pretty(I)) ❸
```

```
      b
      /
      |
      | (0.5*x + sin(x)) dx
      |
      /
      a
```

```
In [113]: int_func = sy.integrate(sy.sin(x) + 0.5 * x, x) ③
```

```
In [114]: print(sy.pretty(int_func)) ③  
          2  
0.25*x - cos(x)
```

```
In [115]: Fb = int_func.subs(x, 9.5).evalf() ④  
          Fa = int_func.subs(x, 0.5).evalf() ④
```

```
In [116]: Fb - Fa ⑤  
Out[116]: 24.3747547180867
```

- ① Объекты `Symbol`, задающие пределы интегрирования.
- ② Определение и вывод объекта `Integral`.
- ③ Нахождение и вывод первообразной.
- ④ Значения первообразных в пределах интегрирования, вычисляемые с помощью методов `subs()` и `evalf()`.
- ⑤ Точное числовое значение интеграла.

Интеграл можно также вычислить символьно в заданных пределах интегрирования.

```
In [117]: int_func_limits = sy.integrate(sy.sin(x) + 0.5 * x,  
                                          (x, a, b)) ①
```

```
In [118]: print(sy.pretty(int_func_limits)) ①  
          2      2  
- 0.25*a + 0.25*b + cos(a) - cos(b)
```

```
In [119]: int_func_limits.subs({a : 0.5, b : 9.5}).evalf() ②  
Out[119]: 24.3747547180868
```

```
In [120]: sy.integrate(sy.sin(x) + 0.5 * x, (x, 0.5, 9.5)) ③  
Out[120]: 24.3747547180867
```

- ① Символьное интегрирование.
- ② Численное интегрирование с подстановкой объекта `dict`.
- ③ Численное интегрирование в один проход.



## Дифференцирование

Производная первообразной равна исходной функции. Проверим это, применив функцию `sy.diff()` к первообразной, записанной в символьном виде.

```
In [121]: int_func.diff()
Out[121]: 0.5*x + sin(x)
```

Как и в примере с интегрированием, мы воспользуемся дифференцированием для поиска точного решения рассмотренной ранее задачи выпуклого программирования. Для этого мы запишем соответствующую функцию в символьном виде, определим ее частные производные и затем найдем корни.

Необходимым, но не достаточным условием глобального минимума функции является равенство ее частных производных нулю. При этом нет никакой гарантии того, что символьное решение будет найдено. Следует учитывать как алгоритмические трудности, так и возможность существования нескольких решений. Но можно решить два уравнения первого порядка в численном виде, предоставив “обоснованные” догадки, которые были получены при решении задач глобальной и локальной минимизации.

```
In [122]: f = (sy.sin(x) + 0.05 * x ** 2
            + sy.sin(y) + 0.05 * y ** 2) ❶
```

```
In [123]: del_x = sy.diff(f, x) ❷
            del_x ❷
Out[123]: 0.1*x + cos(x)
```

```
In [124]: del_y = sy.diff(f, y) ❷
            del_y ❷
Out[124]: 0.1*y + cos(y)
```

```
In [125]: xo = sy.nsolve(del_x, -1.5) ❸
            xo ❸
Out[125]: -1.42755177876459
```

```
In [126]: yo = sy.nsolve(del_y, -1.5) ❸
            yo ❸
Out[126]: -1.42755177876459
```

```
In [127]: f.subs({x : xo, y : yo}).evalf() ❹
Out[127]: -1.77572565314742
```

- ❶ Символьная запись функции.
- ❷ Получение и вывод двух частных производных.
- ❸ Обоснованные догадки о существовании корней и полученные оптимальные значения.
- ❹ Значение глобального минимума функции.

Опять-таки, необоснованные/произвольные предположения могут привести к нахождению локального, а не глобального минимума.

```
In [128]: xo = sy.nsolve(del_x, 1.5) ❶
```

xo

```
Out[128]: 1.74632928225285
```

```
In [129]: yo = sy.nsolve(del_y, 1.5) ❶
```

yo

```
Out[129]: 1.74632928225285
```

```
In [130]: f.subs({x : xo, y : yo}).evalf() ❷
```

```
Out[130]: 2.27423381055640
```

- ❶ Необоснованные предположения о существовании корней.
- ❷ Значение локального минимума функции.

Эти примеры показывают, что равенство нулю частных производных действительно является необходимым, но не достаточным условием существования решения.



### Символьные вычисления

Библиотека SymPy, снабжающая Python возможностью выполнения символьных вычислений, служит ценным инструментом финансовых расчетов. Она особенно полезна при интерактивном финансовом анализе, так как демонстрирует более высокую производительность по сравнению с численными методами.

## Резюме

В этой главе рассматривались математические инструменты, важные с точки зрения финансового анализа. Например, аппроксимация функций применяется во многих областях, таких как построение факторных моделей, интерполяция кривой доходности и регрессионная оценка американских опционов

по методу Монте-Карло. В процессе обработки финансовых данных часто прибегают к выпуклому программированию, когда нужно, например, настроить параметры модели оценки опционов для разных уровней котировок или ожидаемой волатильности.

Численное интегрирование — ключевой инструмент решения многих финансовых задач, таких как оценка опционов и деривативов. После того как получена риск-нейтральная мера вероятности стохастического процесса (или группы процессов), задача ценообразования сводится к определению ожидаемой доходности опциона при такой мере и дисконтированию полученного значения на текущую дату. В главе 12 мы рассмотрим методы моделирования определенных стохастических процессов в риск-нейтральных условиях.

В главе также рассматривались методы символьных вычислений с помощью библиотеки SymPy. Символьные вычисления могут быть очень полезными и эффективными при решении целого ряда математических задач, таких как интегрирование, дифференцирование и решение уравнений.

## Дополнительные ресурсы

Дополнительную информацию о рассмотренных в этой главе библиотеках Python можно найти в Интернете.

- Описание функций NumPy доступно по адресу <https://docs.scipy.org/doc/numpy/reference/>.
- Информация о методах оптимизации и поиске корней уравнений с помощью модуля `scipy.optimize` доступна в документации к библиотеке SciPy (<https://docs.scipy.org/doc/scipy/reference/optimize.html>).
- Функции интегрирования, имеющиеся в модуле `scipy.integrate`, описаны по адресу <https://docs.scipy.org/doc/scipy/reference/integrate.html>.
- Документация к библиотеке SymPy доступна на официальном сайте ([www.sympy.org](http://www.sympy.org)).

Весь необходимый математический аппарат рассмотрен в следующей книге:

- Brandimarte, Paolo. *Numerical Methods in Finance and Economics: A MATLAB-Based Introduction*, 2nd Edition (2006, Wiley).

## Стохастические методы

Предсказуемость не в том, как все пойдет, а в том, как может идти.

*Рахиль Фарук*

В наши дни стохастический анализ — одна из основных математических дисциплин курса экономики и финансов. На заре современной эры финансовых расчетов, преимущественно в 1970–1980-е годы, основной целью финансовых исследований было создание готовых аналитических решений, например для оценки стоимости опционов в рамках определенной финансовой модели. За последние годы требования к вычислительным инструментам значительно изменились. Участникам рынка важно иметь возможность оценивать не только отдельные финансовые инструменты, но и, к примеру, целые реестры деривативов. Точно так же для получения согласованной оценки рисков по всему финансовому учреждению, будь то стоимость под риском или поправка на кредитный риск, приходится учитывать не только финансовую деятельность самого учреждения, но и обязательства всех его контрагентов. Такие сложные задачи могут быть решены только с привлечением мощных численных методов. Именно поэтому стохастический анализ в целом и метод Монте-Карло приобрели особую важность в финансовых вычислениях.

В главе рассматриваются следующие темы.

### *Случайные числа*

Вначале мы познакомимся с псевдослучайными числами, на которых основано большинство методов финансового моделирования.

### *Моделирование*

В финансовых вычислениях ключевую роль играют две задачи: моделирование случайных величин и случайных процессов.

### *Оценка опционов*

В основном мы будем оценивать два производных финансовых инструмента: *европейский опцион* (на указанную дату) и *американский опцион* (на указанный период времени). Существует также *бермудский опцион* с конечным набором дат исполнения.

## Оценка рисков

Методы моделирования хорошо справляются с оценками рисков, такими как стоимость под риском (VaR) или поправка на кредитный риск (CVA).

## Случайные числа

Во всех примерах главы случайные числа<sup>1</sup> генерируются функциями модуля `numpy.random`.

```
In [1]: import math
        import numpy as np
        import numpy.random as npr ❶
        from pylab import plt, mpl

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

❶ Импорт модуля библиотеки NumPy, отвечающего за генерирование случайных чисел.

Например, функция `rand()` генерирует случайные числа на отрезке  $[0, 1)$  с распределением, которое задается в качестве параметра. Функция возвращает объект `ndarray`, что позволяет легко масштабировать полученные случайные числа на более длинный интервал вещественных значений. В частности, ниже показано, как получить случайные числа в диапазоне  $[a, b) = [5, 10)$ , выполняя масштабирование результатов функции `npr.rand()` как в одном, так и в двух измерениях.

```
In [3]: npr.seed(100) ❶
        np.set_printoptions(precision=4) ❶

In [4]: npr.rand(10) ❷
Out[4]: array([0.5434, 0.2784, 0.4245, 0.8448, 0.0047, 0.1216,
               0.6707, 0.8259, 0.1367, 0.5751])

In [5]: npr.rand(5, 5) ❸
Out[5]: array([[0.8913, 0.2092, 0.1853, 0.1084, 0.2197],
               [0.9786, 0.8117, 0.1719, 0.8162, 0.2741],
               [0.4317, 0.94  , 0.8176, 0.3361, 0.1754],
               [0.3728, 0.0057, 0.2524, 0.7957, 0.0153],
```

---

<sup>1</sup> Для простоты мы будем использовать термин *случайные числа*, но помните о том, что все эти числа — *псевдослучайные*.

```
[0.5988, 0.6038, 0.1051, 0.3819, 0.0365]])
```

```
In [6]: a = 5. ④
        b = 10. ⑤
        npr.rand(10) * (b - a) + a ⑥
Out[6]: array([9.4521, 9.9046, 5.2997, 9.4527, 7.8845, 8.7124,
              8.1509, 7.9092, 5.1022, 6.0501])
```

```
In [7]: npr.rand(5, 5) * (b - a) + a ⑦
Out[7]: array([[7.7234, 8.8456, 6.2535, 6.4295, 9.262 ],
              [9.875 , 9.4243, 6.7975, 7.9943, 6.774 ],
              [6.701 , 5.8904, 6.1885, 5.2243, 7.5272],
              [6.8813, 7.964 , 8.1497, 5.713 , 9.6692],
              [9.7319, 8.0115, 6.9388, 6.8159, 6.0217]])
```

- ① Установка затравочного значения, чтобы пример можно было легко воспроизвести, и задание количества цифр после запятой.
- ② Получение одномерного объекта `ndarray` с равномерным распределением.
- ③ Получение двумерного объекта `ndarray` с равномерным распределением.
- ④ Нижний...
- ⑤ ...и верхний пределы...
- ⑥ ...интервала масштабирования случайных чисел.
- ⑦ Масштабирование случайных чисел в двумерном пространстве.

Функции, применяемые для генерирования простых случайных чисел, описаны в табл. 12.1.

**Таблица 12.1.** Функции генерирования простых случайных чисел

Функция	Параметры	Возвращаемый результат
<code>rand()</code>	<code>d0, d1, ..., dn</code>	Случайные числа из заданного распределения
<code>randn()</code>	<code>d0, d1, ..., dn</code>	Выборка (или выборки) из стандартного нормального распределения
<code>randint()</code>	<code>low[, high, size]</code>	Случайные целые числа из диапазона от <i>low</i> (включая) до <i>high</i> (не включая)
<code>random_integers()</code>	<code>low[, high, size]</code>	Случайные целые числа из диапазона от <i>low</i> до <i>high</i> включительно
<code>random_sample()</code>	<code>[size]</code>	Случайные вещественные числа из полуоткрытого интервала $[0,0; 1,0)$

Функция	Параметры	Возвращаемый результат
<code>random()</code>	<code>[size]</code>	Случайные вещественные числа из полуоткрытого интервала $[0,0; 1,0)$
<code>randf()</code>	<code>[size]</code>	Случайные вещественные числа из полуоткрытого интервала $[0,0; 1,0)$
<code>sample()</code>	<code>[size]</code>	Случайные вещественные числа из полуоткрытого интервала $[0,0; 1,0)$
<code>choice()</code>	<code>a[, size, replace, p]</code>	Случайная выборка элементов из заданного одномерного массива
<code>bytes()</code>	<code>length</code>	Случайные байты данных

Визуализировать результаты, возвращаемые перечисленными функциями, несложно. На рис. 12.1 показаны графики двух непрерывных и двух дискретных распределений, сгенерированных следующим кодом.

```
In [8]: sample_size = 500
        rn1 = npr.rand(sample_size, 3) ❶
        rn2 = npr.randint(0, 10, sample_size) ❷
        rn3 = npr.sample(size=sample_size) ❶
        a = [0, 25, 50, 75, 100] ❸
        rn4 = npr.choice(a, size=sample_size) ❸

In [9]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2,
                                                    ncols=2, figsize=(10, 8))
        ax1.hist(rn1, bins=25, stacked=True)
        ax1.set_title('Функция rand()')
        ax1.set_ylabel('Частота')
        ax2.hist(rn2, bins=25)
        ax2.set_title('Функция randint()')
        ax3.hist(rn3, bins=25)
        ax3.set_title('Функция sample()')
        ax3.set_ylabel('Частота')
        ax4.hist(rn4, bins=25)
        ax4.set_title(' Функция choice()');
```

- ❶ Случайные числа с равномерным распределением.
- ❷ Случайные числа из заданного интервала.
- ❸ Случайная выборка значений из конечного списка.

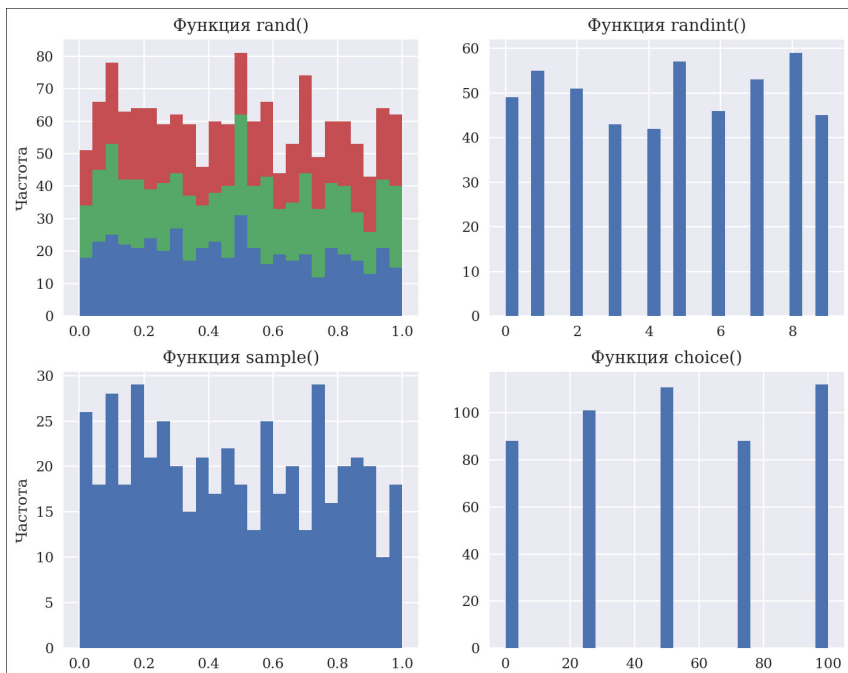


Рис. 12.1. Гистограммы распределения простых случайных чисел

В табл. 12.2 описаны функции, применяемые для генерирования случайных чисел с другими типами распределений.

Таблица 12.2. Функции генерирования случайных чисел, подчиняющихся другим законам распределения

Функция	Параметры	Возвращаемый результат
beta	$a, b[, \text{size}]$	Выборка из бета-распределения в интервале $[0, 1]$
binomial	$n, p[, \text{size}]$	Выборка из биномиального распределения
chisquare	$df[, \text{size}]$	Выборка из распределения хи-квадрат
dirichlet	$\alpha[, \text{size}]$	Выборка из распределения Дирихле
exponential	$[\text{scale}, \text{size}]$	Выборка из экспоненциального распределения
f	$dfnum, dfden[, \text{size}]$	Выборка из распределения Фишера
gamma	$\text{shape}[, \text{scale}, \text{size}]$	Выборка из гамма-распределения
geometric	$p[, \text{size}]$	Выборка из геометрического распределения
gumbel	$[\text{loc}, \text{scale}, \text{size}]$	Выборка из распределения Гумбеля



Функция	Параметры	Возвращаемый результат
hypergeometric	<i>ngood, nbad, nsample[, size]</i>	Выборка из гипергеометрического распределения
laplace	<i>[loc, scale, size]</i>	Выборка из распределения Лапласа, или двойного экспоненциального распределения
logistic	<i>[loc, scale, size]</i>	Выборка из логистического распределения
lognormal	<i>[mean, sigma, size]</i>	Выборка из логнормального распределения
logseries	<i>p[, size]</i>	Выборка из логарифмического распределения
multinomial	<i>n, pvals[, size]</i>	Выборка из мультиномиального распределения
multivariate_normal	<i>mean, cov[, size]</i>	Выборка из многомерного нормального распределения
negative_binomial	<i>n, p[, size]</i>	Выборка из отрицательного биномиального распределения
noncentral_chisquare	<i>df, nonc[, size]</i>	Выборка из асимметричного распределения хи-квадрат
noncentral_f	<i>dfnum, dfden, nonc[, size]</i>	Выборка из асимметричного распределения Фишера
normal	<i>[loc, scale, size]</i>	Выборка из нормального распределения
pareto	<i>a[, size]</i>	Выборка из распределения Парето типа II
poisson	<i>[lam, size]</i>	Выборка из распределения Пуассона
power	<i>a[, size]</i>	Выборка в интервале [0, 1] из степенного распределения с положительной экспонентой $a - 1$
rayleigh	<i>[scale, size]</i>	Выборка из распределения Рэля
standard_cauchy	<i>[size]</i>	Выборка из стандартного распределения Коши с нулевой модой
standard_exponential	<i>[size]</i>	Выборка из стандартного экспоненциального распределения
standard_gamma	<i>shape[, size]</i>	Выборка из стандартного гамма-распределения
standard_normal	<i>[size]</i>	Выборка из стандартного нормального распределения
standard_t	<i>df[, size]</i>	Выборка из распределения Стьюдента с <i>df</i> степенями свободы
triangular	<i>left, mode, right[, size]</i>	Выборка из треугольного распределения в интервале <i>[left, right]</i>
uniform	<i>[low, high, size]</i>	Выборка из равномерного распределения

Функция	Параметры	Возвращаемый результат
<code>vonmises</code>	$\mu$ , $\kappa$ , $size$	Выборка из распределения Мизеса
<code>wald</code>	$mean$ , $scale$ , $size$	Выборка из распределения Вальда, или обратного гауссова распределения
<code>weibull</code>	$a$ , $size$	Выборка из распределения Вейбулла
<code>zipf</code>	$a$ , $size$	Выборка из распределения Ципфа

Несмотря на критику идеи применения нормальных распределений в финансовом анализе, они по-прежнему остаются самыми популярными видами распределений в аналитических приложениях. Одна из причин заключается в том, что многие финансовые модели непосредственно построены на базе нормального или логнормального распределения. А те модели, в которых (лог)нормальное распределение не используется напрямую, всегда можно дискретизировать, а затем аппроксимировать с помощью нормального распределения.

В качестве иллюстрации на рис. 12.2 визуализированы случайные выборки из следующих типов распределений.

- *Стандартное нормальное распределение* с математическим ожиданием  $\mu = 0$  и стандартным отклонением  $\sigma = 1$ .
- *Нормальное распределение* с математическим ожиданием  $\mu = 100$  и стандартным отклонением  $\sigma = 20$ .
- *Распределение хи-квадрат* с 0,5 степенями свободы.
- *Распределение Пуассона* с математическим ожиданием  $\lambda = 1$ .

На рис. 12.2 представлены три непрерывных и одно дискретное распределение (Пуассона). Последнее используется, в частности, для моделирования таких (редких) событий, как скачки стоимости финансового инструмента или обвал рынка.

```
In [10]: sample_size = 500
         rn1 = npr.standard_normal(sample_size) ❶
         rn2 = npr.normal(100, 20, sample_size) ❷
         rn3 = npr.chisquare(df=0.5, size=sample_size) ❸
         rn4 = npr.poisson(lam=1.0, size=sample_size) ❹

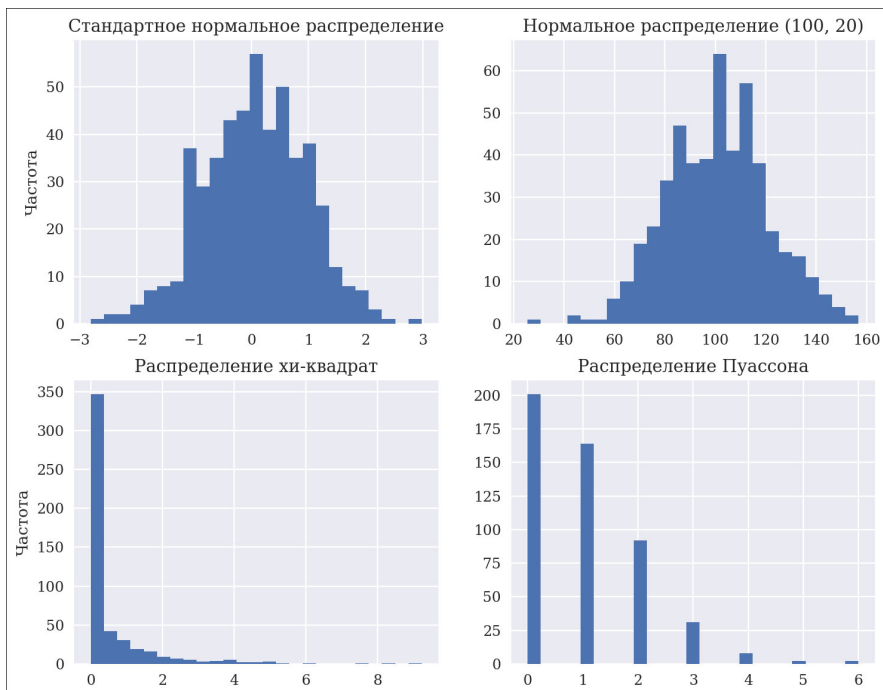
In [11]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2,
                                                    ncols=2, figsize=(10, 8))
```

```

ax1.hist(rn1, bins=25)
ax1.set_title('Стандартное нормальное распределение')
ax1.set_ylabel('Частота')
ax2.hist(rn2, bins=25)
ax2.set_title('Нормальное распределение (100, 20)')
ax3.hist(rn3, bins=25)
ax3.set_title('Распределение хи-квадрат')
ax3.set_ylabel('Частота')
ax4.hist(rn4, bins=25)
ax4.set_title('Распределение Пуассона');

```

- ❶ Случайные числа со стандартным нормальным распределением.
- ❷ Случайные числа с нормальным распределением.
- ❸ Случайные числа с распределением хи-квадрат.
- ❹ Случайные числа с распределением Пуассона.



*Рис. 12.2. Гистограммы случайных выборок из различных распределений*



## Библиотека NumPy и случайные числа

В этом разделе было показано, что библиотека NumPy играет очень важную (порой незаменимую) роль в генерировании псевдослучайных чисел в Python. Создание массивов `ndarray` произвольного размера, заполненных случайными числами, — не только простая, но и эффективная операция.

## Моделирование

*Метод Монте-Карло* — один из самых важных численных методов в финансовом анализе. В основном это обусловлено гибкостью вычисления различных математических выражений (например, интегралов), в том числе финансовых деривативов. Впрочем, гибкость метода достигается за счет достаточно высокой вычислительной нагрузки, поскольку для получения единственной числовой оценки зачастую требуется выполнить сотни тысяч или даже миллионы сложных вычислительных операций.

### Случайные переменные

В качестве примера рассмотрим модель ценообразования опционов Блэка — Шоулза — Мертона. Она описывается уравнением 12.1, где  $S_T$  — будущий уровень фондового индекса на дату  $T$ , а  $S_0$  — текущий уровень индекса.

*Уравнение 12.1. Моделирование будущего уровня индекса по методу Блэка — Шоулза — Мертона*

$$S_T = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}z\right),$$

где

- $S_T$  — уровень индекса на будущую дату  $T$ ;
- $r$  — постоянная безрисковая краткосрочная ставка;
- $\sigma$  — постоянная волатильность (равна стандартному отклонению доходности) индекса;
- $z$  — случайная переменная со стандартным нормальным распределением.

Данная финансовая модель параметризуется и моделируется с помощью следующего кода (рис. 12.3).

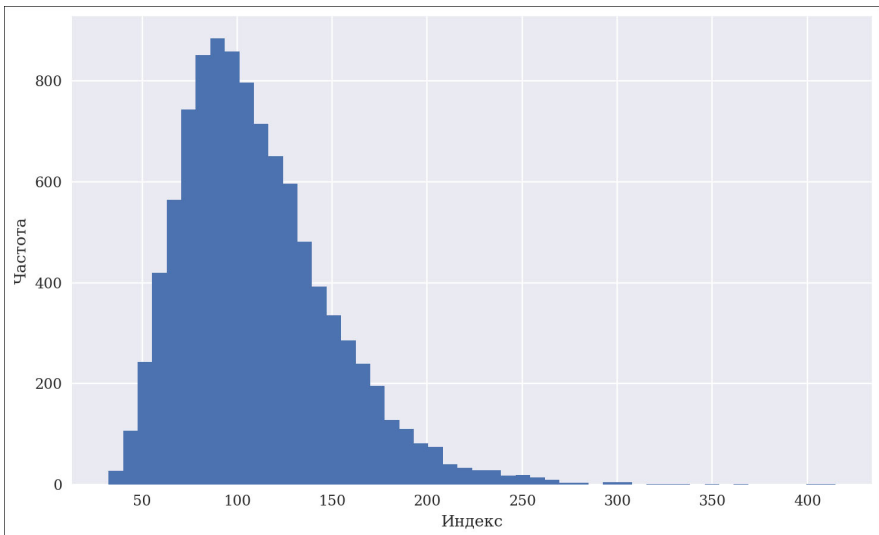
```

In [12]: S0 = 100 ❶
         r = 0.05 ❷
         sigma = 0.25 ❸
         T = 2.0 ❹
         I = 10000 ❺
         ST1 = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                           sigma * math.sqrt(T) * npr.standard_normal(I)) ❻

In [13]: plt.figure(figsize=(10, 6))
         plt.hist(ST1, bins=50)
         plt.xlabel('Индекс')
         plt.ylabel('Частота');

```

- ❶ Начальный уровень индекса.
- ❷ Постоянная безрисковая краткосрочная ставка.
- ❸ Постоянная волатильность.
- ❹ Срок в долях года.
- ❺ Количество траекторий.
- ❻ Моделирование выполняется векторизованным способом. Для дискретизации применяется функция `npr.standard_normal()`.



*Рис. 12.3. Результат статического моделирования геометрического броуновского движения (с помощью функции `npr.standard_normal()`)*

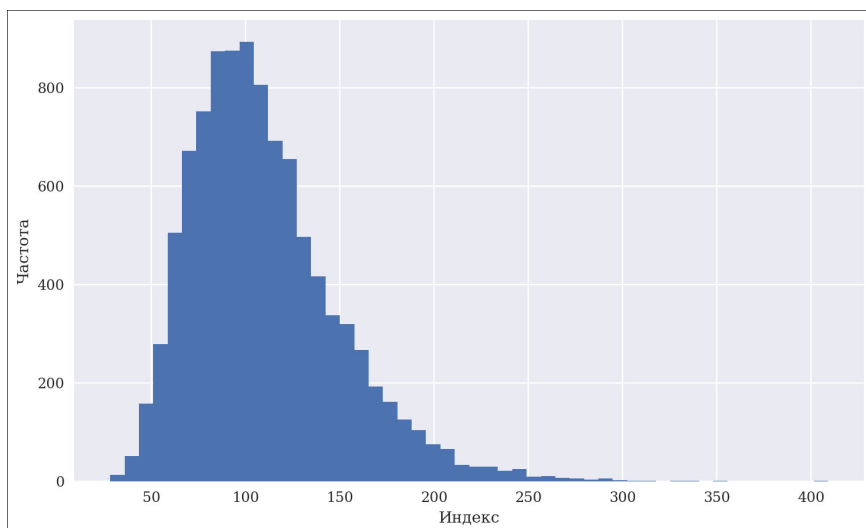
Из рис. 12.3 следует, что случайная величина, определяемая уравнением 12.1, имеет *логнормальное* распределение. Следовательно, можно попробовать непосредственно получить ее с помощью функции `npr.lognormal()`, передав последней значения математического ожидания и стандартного отклонения.

```
In [14]: ST2 = S0 * npr.lognormal((r - 0.5 * sigma ** 2) * T,  
                                sigma * math.sqrt(T), size=I) ❶
```

```
In [15]: plt.figure(figsize=(10, 6))  
plt.hist(ST2, bins=50)  
plt.xlabel('Индекс')  
plt.ylabel('Частота');
```

- ❶ Моделирование выполняется векторизованным способом. Для дискретизации применяется функция `npr.lognormal()`.

Результат моделирования приведен на рис. 12.4.



*Рис. 12.4. Результат статистического моделирования геометрического броуновского движения (с помощью функции `npr.lognormal()`)*

Рис. 12.3 и 12.4 выглядят очень похожими. Можно выполнить более строгую проверку, сравнив моменты двух распределений. Для этого мы воспользуемся модулем `scipy.stats`, определив вспомогательную функцию `print_statistics()`, как показано ниже.

```
In [16]: import scipy.stats as scs
```

```
In [17]: def print_statistics(a1, a2):  
    ''' Вывод статистических показателей  
  
    Параметры  
    =====  
    a1, a2: массивы ndarray  
            содержат результаты моделирования  
    ...  
    sta1 = scs.describe(a1) ❶  
    sta2 = scs.describe(a2) ❶  
    print('%14s %14s %14s' %  
          ('Статистика', 'Набор 1', 'Набор 2'))  
    print(45 * "-")  
    print('%14s %14.3f %14.3f' % ('size', sta1[0],  
                                  sta2[0]))  
    print('%14s %14.3f %14.3f' % ('min', sta1[1][0],  
                                  sta2[1][0]))  
    print('%14s %14.3f %14.3f' % ('max', sta1[1][1],  
                                  sta2[1][1]))  
    print('%14s %14.3f %14.3f' % ('mean', sta1[2],  
                                  sta2[2]))  
    print('%14s %14.3f %14.3f' % ('std', np.sqrt(sta1[3]),  
                                  np.sqrt(sta2[3])))  
    print('%14s %14.3f %14.3f' % ('skew', sta1[4],  
                                  sta2[4]))  
    print('%14s %14.3f %14.3f' % ('kurtosis', sta1[5],  
                                  sta2[5]))
```

```
In [18]: print_statistics(ST1, ST2)
```

Статистика	Набор 1	Набор 2
size	10000.000	10000.000
min	32.327	28.230
max	414.825	409.110
mean	110.730	110.431
std	40.300	39.878
skew	1.122	1.115
kurtosis	2.438	2.217

- ❶ Функция `scs.describe()` возвращает статистические характеристики набора данных.

Как видим, статистические характеристики результатов моделирования оказываются весьма схожими. Наблюдаемые различия в основном обусловлены тем, что в моделировании называется *ошибкой выборки*. Другой тип ошибок, возникающих при *дискретном* моделировании *непрерывных* случайных процессов, — *ошибки дискретизации*. В данном случае они никак не проявляются из-за статической природы процесса моделирования.

## Случайные процессы

Грубо говоря, *случайный (стохастический) процесс* представляет собой последовательность случайных величин. Соответственно, можно предположить, что моделирование подобных процессов будет заключаться в последовательном повторном моделировании одной случайной величины. В основном так и происходит, за тем лишь исключением, что исходы, как правило, не являются независимыми, а зависят от результатов предыдущих исходов.

Но в целом случайные процессы, встречающиеся в финансовых вычислениях, проявляют *марковское свойство*, означающее, что следующее состояние процесса зависит только от его текущего состояния, но не от каких-либо предыдущих состояний или всей цепочки состояний. Такой процесс называется *марковским*.

## Геометрическое броуновское движение

Рассмотрим модель Блэка — Шоулза — Мертона в динамическом виде, который описывается стохастическим дифференциальным уравнением 12.2, где  $Z_t$  — стандартное броуновское движение. Такое уравнение описывает *геометрическое броуновское движение* (geometric Brownian motion — GBM). Значения  $S_t$  подчинены логнормальному распределению, а маржинальная доходность  $\frac{dS_t}{S_t}$  — нормальному распределению.

**Уравнение 12.2. Стохастическое дифференциальное уравнение в модели Блэка — Шоулза — Мертона**

$$dS_t = rS_t dt + \sigma S_t dZ_t.$$

Для точной дискретизации уравнения 12.2 можно воспользоваться методом Эйлера, который представлен уравнением 12.3, где  $\Delta t$  — постоянный шаг дискретизации, а  $z_t$  — случайная величина со стандартным нормальным распределением.



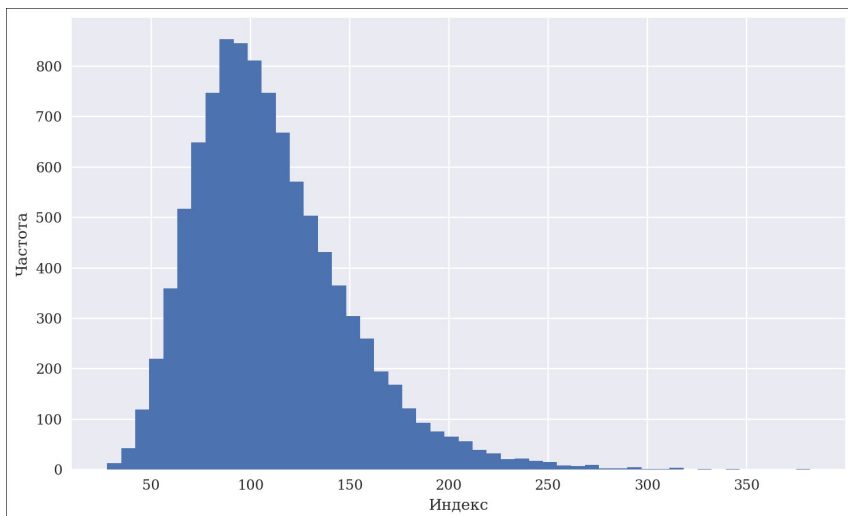
**Уравнение 12.3. Динамическое моделирование индекса в модели Блэка — Шоулза — Мертона**

$$S_t = S_{t-\Delta t} \exp\left(\left(r - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t}z_t\right).$$

Как и в предыдущих случаях, реализация такой формулы в коде Python не вызывает особых трудностей благодаря библиотеке NumPy. Возвращаемые уровни индекса подчиняются логнормальному распределению, как следует из рис. 12.5. Четыре первых момента почти точно повторяют результаты, получаемые при статическом моделировании.

```
In [19]: I = 10000 ❶  
         M = 50 ❷  
         dt = T / M ❸  
         S = np.zeros((M + 1, I)) ❹  
         S[0] = S0 ❺  
         for t in range(1, M + 1):  
             S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt + \  
                                     sigma * math.sqrt(dt) * npr.standard_normal(I)) ❻  
  
In [20]: plt.figure(figsize=(10, 6))  
         plt.hist(S[-1], bins=50)  
         plt.xlabel('Индекс')  
         plt.ylabel('Частота');
```

- ❶ Количество траекторий.
- ❷ Количество временных интервалов дискретизации.
- ❸ Длительность временного интервала в долях года.
- ❹ Двухмерный объект `ndarray` со значениями индекса.
- ❺ Начальные значения в исходный момент времени  $t = 0$ .
- ❻ Моделирование выполняется частично векторизованным способом. Цикл обеспечивает обработку временного интервала начиная с  $t = 1$  и заканчивая  $t = T$ .



*Рис. 12.5. Динамическое моделирование уровня индекса с помощью геометрического броуновского движения*

Ниже приведено сравнение статистических характеристик динамического и статического методов моделирования. На рис. 12.6 приведены результаты моделирования первых 10 траекторий.

In [21]: `print_statistics(S[-1], ST2)`

Статистика	Набор 1	Набор 2
size	10000.000	10000.000
min	27.746	28.230
max	382.096	409.110
mean	110.423	110.431
std	39.179	39.878
skew	1.069	1.115
kurtosis	2.028	2.217

```
In [22]: plt.figure(figsize=(10, 6))
plt.plot(S[:, :10], lw=1.5)
plt.xlabel('Время')
plt.ylabel('Индекс');
```



**Рис. 12.6.** Динамическое моделирование траекторий геометрического броуновского движения

Таким образом, динамическое моделирование позволяет не только визуализировать траектории моделирования, но и оценить опционы американского/бермудского типа или опционы, выплаты по которым основаны на отдельных траекториях. Это дает возможность наблюдать за развитием событий во времени.

### **Диффузия по закону квадратного корня**

Поведение многих финансовых индикаторов, например коротких ставок или волатильности индексов, можно представить с помощью процессов *возврата к среднему* (mean reversion). Для их математического описания широко применяется модель *диффузии по закону квадратного корня*, предложенная Коксом, Ингерсоллом и Россом [2]. Соответствующее стохастическое дифференциальное уравнение имеет следующий вид.

**Уравнение 12.4.** Стохастическое дифференциальное уравнение для диффузии по закону квадратного корня

$$dx_t = k(\theta - x_t)dt + \sigma\sqrt{x_t}dZ_t,$$

где

- $x_t$  — уровень процесса на дату  $t$ ;
- $k$  — коэффициент возврата к среднему;

- $\theta$  — долгосрочное среднее значение процесса;
- $\sigma$  — постоянный коэффициент волатильности;
- $Z_t$  — стандартное броуновское движение.

Общеизвестно, что в таких моделях значения  $x_t$  подчиняются распределению хи-квадрат. Тем не менее, как указывалось ранее, многие финансовые модели можно дискретизировать и аппроксимировать с помощью нормального распределения (метод Эйлера). Для геометрического броуновского движения метод Эйлера дает точные результаты, в большинстве остальных стохастических процессов результаты получатся смещенными. Но даже когда имеется точный метод дискретизации (как в случае рассмотренной далее диффузии по закону квадратного корня), метод Эйлера может оказаться более предпочтительным, исходя из вычислительных соображений. Приняв, что  $s = t - \Delta t$  и  $x^+ \equiv \max(x, 0)$ , получим уравнение 12.5.

**Уравнение 12.5. Метод Эйлера для диффузии по закону квадратного корня**

$$\begin{aligned}\tilde{x}_t &= \tilde{x}_s + k(\theta - \tilde{x}_s^+) \Delta t + \sigma \sqrt{\tilde{x}_s^+} \sqrt{\Delta t} z_t, \\ x_t &= \tilde{x}_t^+.\end{aligned}$$

Диффузия по закону квадратного корня удобна тем, что значения  $x_t$  остаются строго положительными. Это очень важно, поскольку при дискретизации по методу Эйлера исключать отрицательные значения нельзя. Вот почему приходится всегда работать с положительными значениями моделируемого процесса. Реализация такого подхода в коде требует применения сразу двух массивов `ndarray`. Результаты моделирования представлены в виде гистограммы на рис. 12.7.

```
In [23]: x0 = 0.05 ❶
         kappa = 3.0 ❷
         theta = 0.02 ❸
         sigma = 0.1 ❹
         I = 10000
         M = 50
         dt = T / M

In [24]: def srd_euler():
         xh = np.zeros((M + 1, I))
         x = np.zeros_like(xh)
         xh[0] = x0
         x[0] = x0
```

```

for t in range(1, M + 1):
    xh[t] = (xh[t - 1] + \
             kappa * (theta - np.maximum(xh[t - 1], 0)) * dt + \
             sigma * np.sqrt(np.maximum(xh[t - 1], 0)) * \
             math.sqrt(dt) * npr.standard_normal(1)) ❸
    x = np.maximum(xh, 0)
    return x
x1 = srd_euler()

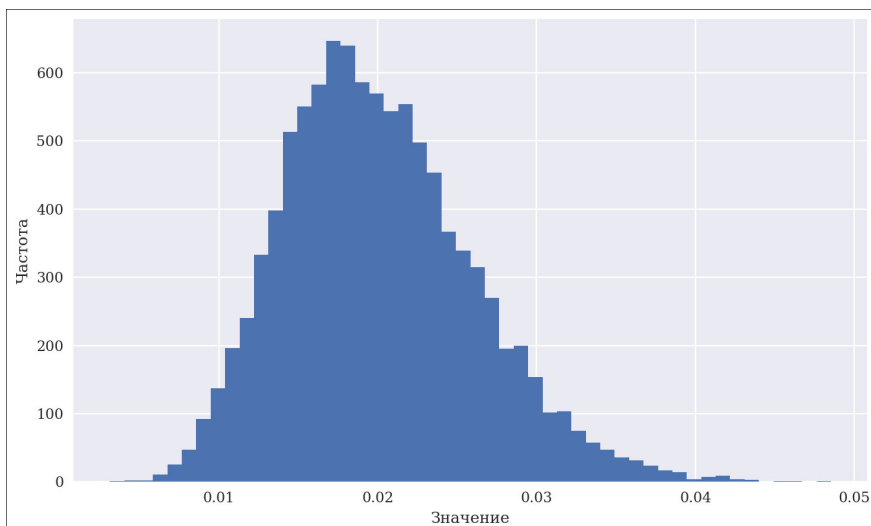
```

```

In [25]: plt.figure(figsize=(10, 6))
plt.hist(x1[-1], bins=50)
plt.xlabel('Значение')
plt.ylabel('Частота');

```

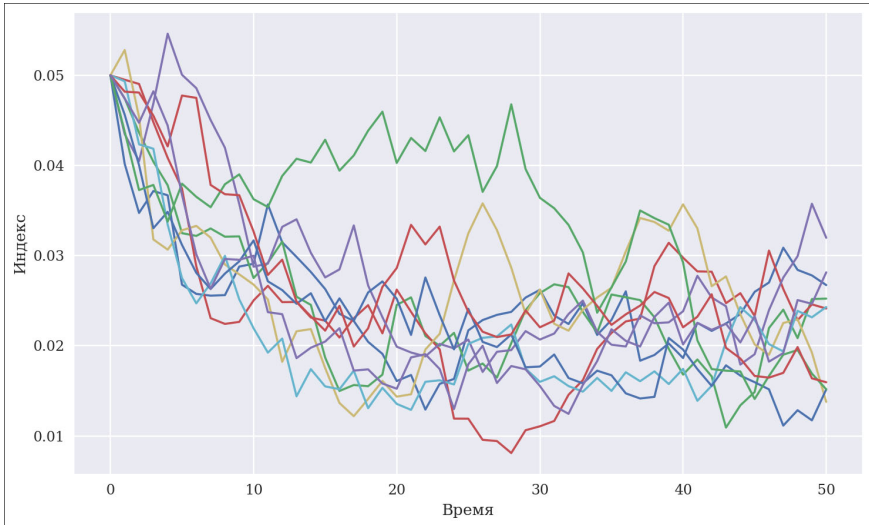
- ❶ Начальное значение (например, короткой ставки).
- ❷ Коэффициент возврата к среднему.
- ❸ Долгосрочное среднее значение.
- ❹ Коэффициент волатильности.
- ❺ Моделирование по методу Эйлера.



*Рис. 12.7. Динамическое моделирование диффузии по закону квадратного корня (метод Эйлера)*

На рис. 12.8 показаны графики первых 10 траекторий процесса, на которых просматривается тенденция к отрицательному дрейфу (поскольку  $x_0 > \theta$ ) и сходимости к точке  $\theta = 0,02$ .

```
In [26]: plt.figure(figsize=(10, 6))
plt.plot(x1[:, :10], lw=1.5)
plt.xlabel('Время')
plt.ylabel('Индекс');
```



*Рис. 12.8. Динамическое моделирование траекторий диффузии по закону квадратного корня (метод Эйлера)*

Уравнение 12.6 описывает точную схему дискретизации для диффузии по закону квадратного корня, основанную на нецентральной распределении хи-квадрат  $\chi_d'^2$  с

$$df = \frac{4\theta k}{\sigma^2}$$

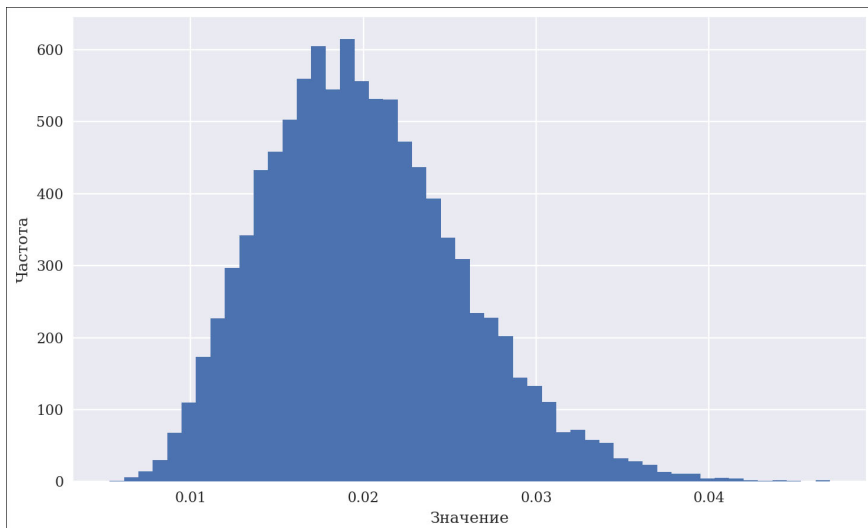
степенями свободы и параметром нецентральности

$$nc = \frac{4ke^{-k\Delta t}}{\sigma^2(1 - e^{-k\Delta t})} x_s.$$

**Уравнение 12.6. Точная схема дискретизации для диффузии по закону квадратного корня**

$$x_t = \frac{\sigma^2 (1 - e^{-k\Delta t})}{4k} \chi_d^2 \left( \frac{4ke^{-k\Delta t}}{\sigma^2 (1 - e^{-k\Delta t})} x_s \right).$$

Реализация такого уравнения на Python получится чуть более сложной, но все равно достаточно компактной. Результат в виде гистограммы показан на рис. 12.9.



**Рис. 12.9.** Динамическое моделирование диффузии по закону квадратного корня (точная схема)

```
In [27]: def srd_exact():
    x = np.zeros((M + 1, I))
    x[0] = x0
    for t in range(1, M + 1):
        df = 4 * theta * kappa / sigma ** 2 ❶
        c = (sigma ** 2 * (1 - np.exp(-kappa * dt))) /
            (4 * kappa) ❷
        nc = np.exp(-kappa * dt) / c * x[t - 1] ❸
        x[t] = c * npr.noncentral_chisquare(df, nc, size=I) ❹
    return x
x2 = srd_exact()
```

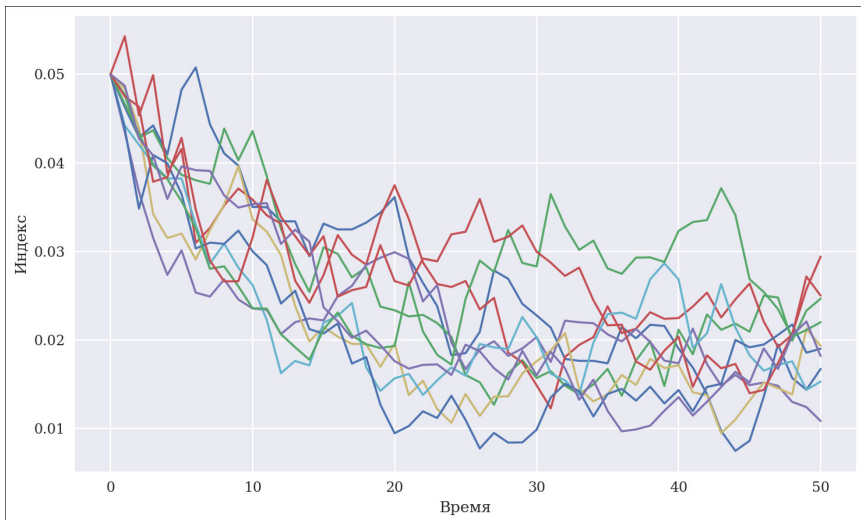
```
In [28]: plt.figure(figsize=(10, 6))
```

```
plt.hist(x2[-1], bins=50)
plt.xlabel('Значение')
plt.ylabel('Частота');
```

- ❶ Точная схема дискретизации на основе функции `prg.noncentral_chisquare()`.

На рис. 12.10 показаны графики первых 10 траекторий процесса, на которых точно так же просматривается тенденция к отрицательному дрейфу и схождению значений к  $\theta$ .

```
In [29]: plt.figure(figsize=(10, 6))
plt.plot(x2[:, :10], lw=1.5)
plt.xlabel('Время')
plt.ylabel('Индекс');
```



*Рис. 12.10. Динамическое моделирование траекторий диффузии по закону квадратного корня (точная схема)*

Сравнение основных статистических характеристик обеих схем позволяет сделать вывод о том, что смещенный метод Эйлера дает достаточно хорошие результаты с точки зрения статистики.

```
In [30]: print_statistics(x1[-1], x2[-1])
```

Статистика	Набор 1	Набор 2
size	10000.000	10000.000
min	0.003	0.005



max	0.049	0.047
mean	0.020	0.020
std	0.006	0.006
skew	0.529	0.532
kurtosis	0.289	0.273

```
In [31]: I = 250000
         %time x1 = srd_euler()
         CPU times: user 1.62 s, sys: 184 ms, total: 1.81 s
         Wall time: 1.08 s
```

```
In [32]: %time x2 = srd_exact()
         CPU times: user 3.29 s, sys: 39.8 ms, total: 3.33 s
         Wall time: 1.98 s
```

```
In [33]: print_statistics(x1[-1], x2[-1])
         x1 = 0.0; x2 = 0.0
```

Статистика	Набор 1	Набор 2
size	250000.000	250000.000
min	0.002	0.003
max	0.071	0.055
mean	0.020	0.020
std	0.006	0.006
skew	0.563	0.579
kurtosis	0.492	0.520

В то же время можно заметить существенную разницу с точки зрения скорости вычислений, поскольку выборка из нецентрального распределения хи-квадрат — намного более затратный процесс, чем выборка из стандартного нормального распределения. Моделирование по точной схеме отнимает вдвое больше времени, чем моделирование по методу Эйлера при практически идентичных результатах.

## Стохастическая волатильность

Главное допущение модели Блэка — Шоулза — Мертона заключается в *постоянстве* волатильности. В действительности волатильность не является ни постоянной, ни детерминированной величиной — она имеет абсолютно *стохастический* характер. Поэтому прорыв в финансовом моделировании произошел только в начале 1990-х годов с появлением так называемых *моделей стохастической волатильности*. Одна из самых популярных моделей такого типа — модель Хестона [3], описываемая уравнением 12.7.

**Уравнение 12.7. Стохастические дифференциальные уравнения для модели стохастической волатильности Хестона**

$$\begin{aligned}dS_t &= rS_t dt + \sqrt{v_t} S_t dZ_t^1, \\dv_t &= k_v (\theta_v - v_t) dt + \sigma_v \sqrt{v_t} dZ_t^2, \\dZ_t^1 dZ_t^2 &= \rho.\end{aligned}$$

Назначение переменных и параметров здесь то же самое, что и в формулах, описывающих геометрическое броуновское движение и диффузию по закону квадратного корня. Параметр  $\rho$  определяет мгновенную корреляцию между двумя стандартными броуновскими движениями  $Z_t^1$  и  $Z_t^2$ . Такой подход позволяет учитывать эффект леввериджа, суть которого заключается в том, что волатильность растет в период кризиса (сужающийся рынок) и снижается в период процветания (растущий рынок).

Рассмотрим следующий вариант параметризации модели. Чтобы учесть корреляцию между двумя стохастическими процессами, необходимо выполнить разложение Холецкого для матрицы корреляционных коэффициентов.

In [34]: `S0 = 100.`

`r = 0.05`

`v0 = 0.1` ❶

`kappa = 3.0`

`theta = 0.25`

`sigma = 0.1`

`rho = 0.6` ❷

`T = 1.0`

In [35]: `corr_mat = np.zeros((2, 2))`

`corr_mat[0, :] = [1.0, rho]`

`corr_mat[1, :] = [rho, 1.0]`

`cho_mat = np.linalg.cholesky(corr_mat)` ❸

In [36]: `cho_mat` ❸

Out[36]: `array([[1. , 0. ],  
[0.6, 0.8]])`

- ❶ Исходное (моментальное) значение волатильности.
- ❷ Постоянная корреляция между двумя броуновскими движениями.
- ❸ Разложение Холецкого и результирующая матрица.

Перед началом моделирования случайных процессов для них генерируются полные наборы случайных чисел: набор 0 применяется при моделировании индексов, а набор 1 — при моделировании волатильности. Во втором случае для расчета диффузии по закону квадратного корня применяется метод Эйлера, учитывающий корреляцию с помощью матрицы Холецкого.

```
In [37]: M = 50
         I = 10000
         dt = T / M

In [38]: ran_num = npr.standard_normal((2, M + 1, I)) ❶

In [39]: v = np.zeros_like(ran_num[0])
         vh = np.zeros_like(v)

In [40]: v[0] = v0
         vh[0] = v0

In [41]: for t in range(1, M + 1):
         ran = np.dot(cho_mat, ran_num[:, t, :]) ❷
         vh[t] = (vh[t - 1] +
                  kappa * (theta - np.maximum(vh[t - 1], 0)) * dt +
                  sigma * np.sqrt(np.maximum(vh[t - 1], 0)) *
                  math.sqrt(dt) * ran[1]) ❸

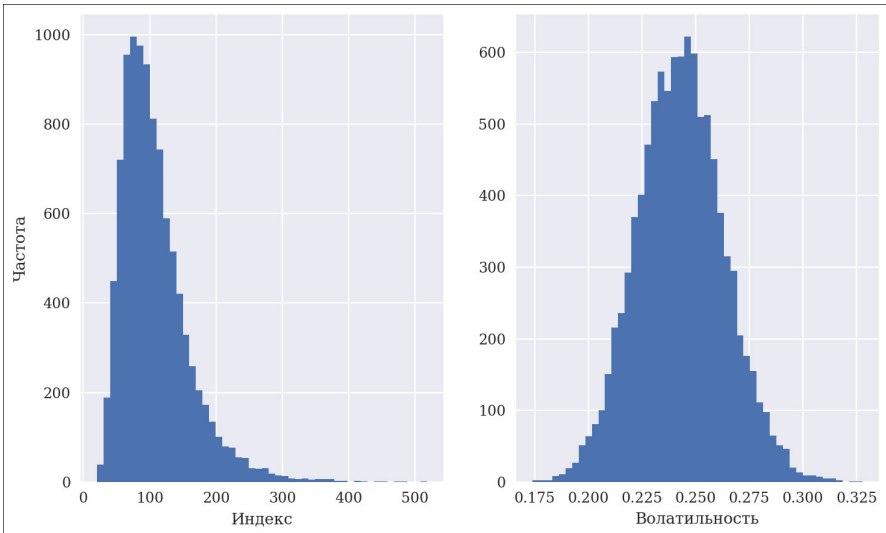
In [42]: v = np.maximum(vh, 0)
```

- ❶ Генерирование трехмерного набора случайных чисел.
- ❷ Выбор подходящего поднабора случайных чисел и его преобразование с помощью матрицы Холецкого.
- ❸ Отдельные траектории, рассчитанные по методу Эйлера.

При моделировании индексов тоже учитывается корреляция, но в данном случае применяется точная схема для геометрического броуновского движения. На рис. 12.11 показаны результаты моделирования как для уровня индекса, так и для волатильности.

```
In [43]: S = np.zeros_like(ran_num[0])
         S[0] = S0
         for t in range(1, M + 1):
             ran = np.dot(cho_mat, ran_num[:, t, :])
             S[t] = S[t - 1] * np.exp((r - 0.5 * v[t]) * dt + \
                                     np.sqrt(v[t]) * ran[0] * np.sqrt(dt))
```

```
In [44]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))
ax1.hist(S[-1], bins=50)
ax1.set_xlabel('Индекс')
ax1.set_ylabel('Частота')
ax2.hist(v[-1], bins=50)
ax2.set_xlabel('Волатильность');
```



**Рис. 12.11.** Динамическое моделирование стохастической волатильности

В данном примере наглядно продемонстрировано еще одно преимущество применения метода Эйлера для расчета диффузии по закону квадратного корня: *простота и последовательность учета корреляции для наборов случайных чисел, подчиняющихся стандартному нормальному распределению*. Подобное невозможно в комбинированном подходе, когда метод Эйлера используется только в модели оценки индексов, а моделирование волатильности выполняется по точной схеме согласно нецентральному распределению хи-квадрат.

Изучив графики первых 10 траекторий каждого процесса (рис. 12.12), можно сделать вывод о том, что волатильность характеризуется положительным дрейфом к среднему и вполне ожидаемо сходится к значению  $\theta = 0,25$ .

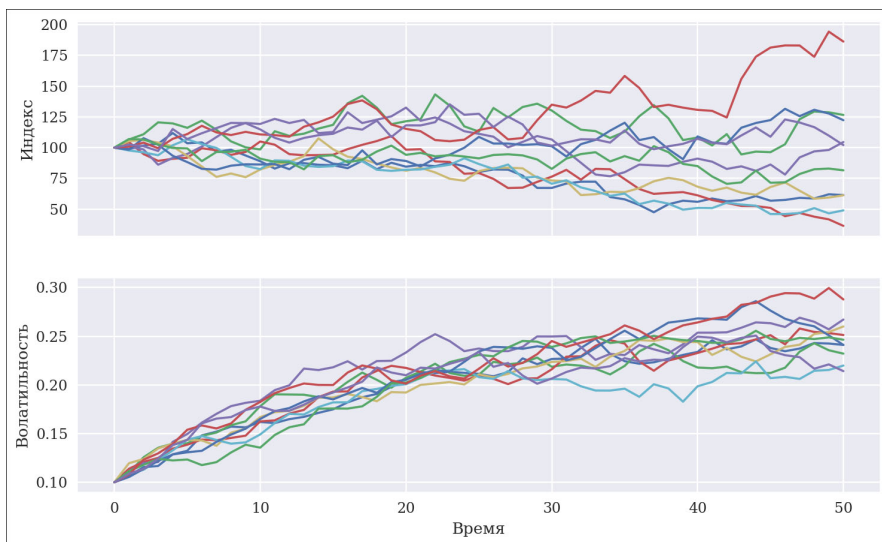
```
In [45]: print_statistics(S[-1], v[-1])
```

Статистика	Набор 1	Набор 2
size	10000.000	10000.000
min	20.556	0.174

max	517.798	0.328
mean	107.843	0.243
std	51.341	0.020
skew	1.577	0.124
kurtosis	4.306	0.048

```
In [46]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
figsize=(10, 6))
```

```
ax1.plot(S[:, :10], lw=1.5)
ax1.set_ylabel('Индекс')
ax2.plot(v[:, :10], lw=1.5)
ax2.set_xlabel('Время')
ax2.set_ylabel('Волатильность');
```



*Рис. 12.12. Отдельные траектории, полученные при моделировании стохастической волатильности*

Быстрый анализ статистических характеристик обоих наборов данных показывает, что индекс характеризуется чрезвычайно высоким максимальным значением. В действительности оно намного выше, чем можно получить при моделировании процесса с помощью геометрического броуновского движения (при прочих равных условиях).

## Прыжковая диффузия

Стохастичность волатильности и эффект левериджа — это эмпирические наблюдения, которые находят подтверждение на большом количестве рынков. Другой важный эмпирический факт — существование скачков в стоимости активов и волатильности. В 1976 году Мертоном [6] была описана модель прыжковой диффузии (jump diffusion — JD), дополняющая формулу Блэка — Шоулза — Мертона компонентом, который учитывает скачки в логнормальном распределении. Риск-нейтральное стохастическое дифференциальное уравнение обновленной модели приведено ниже.

**Уравнение 12.8.** Стохастическое дифференциальное уравнение для модели прыжковой диффузии Мертона

$$dS_t = (r - r_J) S_t dt + \sigma S_t dZ_t + J_t S_t dN_t.$$

Для полноты еще раз приведем описание всех параметров и переменных:

- $S_t$  — уровень индекса на дату  $t$ ;
- $r$  — постоянная безрисковая краткосрочная ставка;
- $r_J \equiv \lambda \left( e^{\mu_J + \delta^2/2} - 1 \right)$  — коррекция дрейфа при скачках для обеспечения риск-нейтральности;
- $\sigma$  — постоянная волатильность величины  $S$ ;
- $Z_t$  — стандартное броуновское движение;
- $J_t$  — скачок на дату  $t$  с распределением...

$$\dots \log(1 + J_t) \approx N \left( \log(1 + \mu_J) - \frac{\delta^2}{2}, \delta^2 \right), \text{ где } \dots$$

... $N$  — кумулятивная функция распределения случайной величины со стандартным нормальным распределением;

- $N_t$  — пуассоновский процесс с интенсивностью  $\lambda$ .

В уравнении 12.9 приведена формула дискретизации Эйлера для прыжковой диффузии, где величины  $z_t^n$  подчинены стандартному нормальному распределению, а  $y_t$  — распределению Пуассона с интенсивностью  $\lambda$ .

**Уравнение 12.9.** Дискретизация по методу Эйлера для модели прыжковой диффузии

$$S_t = S_{t-\Delta t} \left( e^{(r-r_J-\sigma^2/2)\Delta t + \sigma\sqrt{\Delta t}z_t^n} + \left( e^{\mu_J + \delta z_t^2} - 1 \right) y_t \right).$$

Чтобы реализовать такую схему дискретизации, требуется выполнить численную параметризацию модели.

```
In [47]: S0 = 100.  
         r = 0.05  
         sigma = 0.2  
         lamb = 0.75 ❶  
         mu = -0.6 ❷  
         delta = 0.25 ❸  
         rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1) ❹
```

```
In [48]: T = 1.0  
         M = 50  
         I = 10000  
         dt = T / M
```

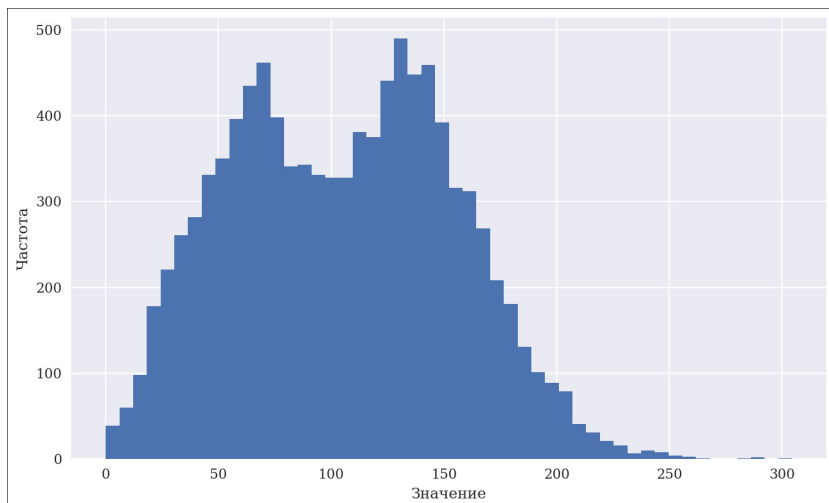
- ❶ Интенсивность скачка.
- ❷ Средняя величина скачка.
- ❸ Волатильность скачков.
- ❹ Коррекция дрейфа.

На этот раз нам понадобятся сразу три набора случайных чисел. Обратите внимание на второй пик (бимодальное частотное распределение) на гистограмме, показанной на рис. 12.13, который возник вследствие скачка.

```
In [49]: S = np.zeros((M + 1, I))  
         S[0] = S0  
         sn1 = npr.standard_normal((M + 1, I)) ❶  
         sn2 = npr.standard_normal((M + 1, I)) ❶  
         poi = npr.poisson(lamb * dt, (M + 1, I)) ❷  
         for t in range(1, M + 1, 1):  
             S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * sigma ** 2) * \  
                                     dt + sigma * math.sqrt(dt) * sn1[t]) + \  
                               (np.exp(mu + delta * sn2[t]) - 1) * poi[t]) ❸  
             S[t] = np.maximum(S[t], 0)
```

```
In [50]: plt.figure(figsize=(10, 6))  
         plt.hist(S[-1], bins=50)  
         plt.xlabel('Значение')  
         plt.ylabel('Частота');
```

- ❶ Случайные числа со стандартным нормальным распределением.
- ❷ Случайные числа с распределением Пуассона.
- ❸ Моделирование на основе точного метода Эйлера.



*Рис. 12.13. Динамическое моделирование прыжковой диффузии*

На первых 10 траекториях можно также заметить отрицательные скачки, как показано на рис. 12.14.

```
In [51]: plt.figure(figsize=(10, 6))
plt.plot(S[:, :10], lw=1.5)
plt.xlabel('Время')
plt.ylabel('Индекс');
```



*Рис. 12.14. Динамическое моделирование траекторий прыжковой диффузии*



## Уменьшение дисперсии

Все применявшиеся ранее функции Python позволяют генерировать только *псевдослучайные* числа, а поскольку размеры выборок меняются, получаемые наборы чисел далеко не всегда обладают ожидаемыми статистическими характеристиками. Предположим, требуется получить набор случайных чисел со стандартным нормальным распределением, в котором математическое ожидание равно 0, а стандартное отклонение — 1. Проверим, какие статистические характеристики будут у различных наборов случайных чисел. Для более реалистичного сравнения зафиксируем затравочное значение генератора случайных чисел.

```
In [52]: print('%15s %15s' % ('Среднее', 'Стандартное отклонение'))
print(31 * '-')
for i in range(1, 31, 2):
    npr.seed(100)
    sn = npr.standard_normal(i ** 2 * 10000)
    print('%15.12f %15.12f' % (sn.mean(), sn.std()))
        Среднее      Стандартное отклонение
-----
0.001150944833      1.006296354600
0.002841204001      0.995987967146
0.001998082016      0.997701714233
0.001322322067      0.997771186968
0.000592711311      0.998388962646
-0.000339730751      0.998399891450
-0.000228109010      0.998657429396
0.000295768719      0.998877333340
0.000257107789      0.999284894532
-0.000357870642      0.999456401088
-0.000528443742      0.999617831131
-0.000300171536      0.999445228838
-0.000162924037      0.999516059328
0.000135778889      0.999611052522
0.000182006048      0.999619405229
```

```
In [53]: i ** 2 * 10000
Out[53]: 8410000
```

Полученные результаты свидетельствуют о том, что статистические показатели улучшаются с увеличением количества исходов<sup>2</sup>. Но они все еще дале-

---

<sup>2</sup> Здесь действует закон больших чисел.

ки от требуемых даже в самой большой выборке, которая насчитывает свыше 8 000 000 случайных чисел.

К счастью, существуют простые и универсальные способы уменьшения дисперсии, позволяющие уравновесить два первых момента (стандартного) нормального распределения. Один из них — *метод симметричных выборок* (antithetic variates). В этом методе берется только половина от требуемого количества случайных значений, а затем к ним добавляется точно такой же набор, но с обратным знаком<sup>3</sup>. Например, если генератор случайных чисел (т.е. соответствующая функция Python) возвращает число 0.5, то помимо него в конечный набор включается также значение -0.5. В результате среднее такого набора будет равно нулю.

В NumPy данный подход к получению случайных чисел реализует функция `np.concatenate()`. Следующий пример повторяет предыдущий, только теперь в нем используются симметричные выборки.

```
In [54]: sn = npr.standard_normal(int(10000 / 2))
         sn = np.concatenate((sn, -sn)) ❶
```

```
In [55]: np.shape(sn) ❷
Out[55]: (10000,)
```

```
In [56]: sn.mean() ❸
Out[56]: 2.842170943040401e-18
```

```
In [57]: print('%15s %15s' % ('Среднее', 'Стандартное отклонение'))
         print(31 * "-")
         for i in range(1, 31, 2):
             npr.seed(1000)
             sn = npr.standard_normal(i ** 2 * int(10000 / 2))
             sn = np.concatenate((sn, -sn))
             print("%15.12f %15.12f" % (sn.mean(), sn.std()))
             Среднее      Стандартное отклонение
         -----
         0.00000000000000      1.009653753942
         -0.00000000000000      1.000413716783
         0.00000000000000      1.002925061201
         -0.00000000000000      1.000755212673
         0.00000000000000      1.001636910076
         -0.00000000000000      1.000726758438
```

---

<sup>3</sup> Описанный метод работает только для симметричных случайных величин с нулевой медианой, как в стандартном нормальном распределении.

-0.000000000000	1.001621265149
0.000000000000	1.001203722778
-0.000000000000	1.00055669784
-0.000000000000	1.000113464185
-0.000000000000	0.999435175324
-0.000000000000	0.999356961431
-0.000000000000	0.999641436845
-0.000000000000	0.999642768905
-0.000000000000	0.999638303451

- ❶ Конкатенация двух массивов `ndarray`...
- ❷ ...для получения нужного количества случайных чисел.
- ❸ Среднее результирующего набора равно нулю (в пределах стандартной арифметической ошибки округления чисел с плавающей точкой).

Как видите, метод симметричных выборок позволяет получить идеальный первый момент для набора случайных чисел, что и не удивительно. Однако он не оказывает никакого влияния на второй момент — стандартное отклонение. Существует другой способ понижения дисперсии — *метод моментов* (moment matching), с помощью которого можно за один шаг улучшить как первый, так и второй момент.

```
In [58]: sn = npr.standard_normal(10000)
```

```
In [59]: sn.mean()
```

```
Out[59]: -0.001165998295162494
```

```
In [60]: sn.std()
```

```
Out[60]: 0.991255920204605
```

```
In [61]: sn_new = (sn - sn.mean()) / sn.std() ❶
```

```
In [62]: sn_new.mean()
```

```
Out[62]: -2.3803181647963357e-17
```

```
In [63]: sn_new.std()
```

```
Out[63]: 0.9999999999999999
```

- ❶ Одновременная коррекция первого и второго моментов.

Если мы вычтем среднее из каждого случайного числа и разделим результат на стандартное отклонение, то добьемся почти идеального соответствия первому и второму моментам стандартного нормального распределения.

Описанные методы понижения дисперсии реализованы в следующей функции, которая генерирует случайные числа со стандартным нормальным распределением.

```
In [64]: def gen_sn(M, I, anti_paths=True, mo_match=True):
        ''' Функция, генерирующая случайные числа
            для моделирования.

            Параметры
            =====
            M: int
                Количество временных интервалов дискретизации
            I: int
                Количество моделируемых траекторий
            anti_paths: булево значение
                Применение симметричных выборок
            mo_match: булево значение
                Применение метода моментов
            ...
            if anti_paths is True:
                sn = npr.standard_normal((M + 1, int(I / 2)))
                sn = np.concatenate((sn, -sn), axis=1)
            else:
                sn = npr.standard_normal((M + 1, I))
            if mo_match is True:
                sn = (sn - sn.mean()) / sn.std()
            return sn
```



## Векторизация и моделирование

Векторизация вычислений с помощью инструментов NumPy — удобный и эффективный способ реализации метода Монте-Карло в Python. В то же время он оказывается очень требовательным к ресурсам памяти. Альтернативные решения, обеспечивающие аналогичную производительность, рассматривались в главе 10.

## Оценка опционов

Одно из наиболее важных применений метода Монте-Карло заключается в оценке *условных требований* (опционов, фьючерсов, гибридных инструментов и т.п.). Если объяснять простым языком, то в риск-нейтральных процессах стоимость условного требования определяется как дисконтированная ожидаемая

выплата в риск-нейтральной (мартингальной) мере. Это мера вероятности того, что все факторы риска (акции, индексы и пр.) будут дрейфовать с безрисковой краткосрочной ставкой, делая процессы дисконтирования мартингалами<sup>4</sup>. Согласно *фундаментальной теореме ценообразования финансовых активов* существование такой меры вероятности равнозначно отсутствию арбитража.

В финансах опцион — это право на покупку (колл) или продажу (пут) определенного финансового инструмента при наступлении определенной даты (*европейский опцион*) или в течение заданного периода времени (*американский опцион*) за оговоренную цену (*цена исполнения, или страйк-цена*). Сначала мы рассмотрим самый простой случай — ценообразование европейского опциона.

## Европейские опционы

Выплаты по европейскому колл-опциону для фондового индекса на дату исполнения рассчитываются по формуле  $h(S_T) \equiv \max(S_T - K, 0)$ , где  $S_T$  — уровень индекса на дату исполнения  $T$ , а  $K$  — цена исполнения (страйк-цена). С учетом риск-нейтральной (на полных рынках) меры для соответствующего случайного процесса (например, геометрического броуновского движения) стоимость такого опциона определяется по формуле, описываемой уравнением 12.10.

**Уравнение 12.10.** *Ценообразование опциона в риск-нейтральных условиях*

$$C_0 = e^{-rT} \mathbb{E}_0^Q(h(S_T)) = e^{-rT} \int_0^\infty h(s)q(s)ds.$$

Принципы численного интегрирования по методу Монте-Карло были рассмотрены в предыдущей главе. Попробуем применить этот подход к уравнению 12.10. Уравнение 12.11 позволяет получить оценку стоимости европейского опциона по методу Монте-Карло, где  $\tilde{S}_T^i$  —  $T$ -й смоделированный уровень индекса на дату исполнения.

**Уравнение 12.11.** *Риск-нейтральная оценка стоимости опциона по методу Монте-Карло*

$$\tilde{C}_0 = e^{-rT} \frac{1}{I} \sum_{i=1}^I h(\tilde{S}_T^i).$$

<sup>4</sup> О мартингалах можно прочитать в Википедии (<https://ru.wikipedia.org/wiki/Мартингал>).

Теперь можно переходить к параметризации модели геометрического броуновского движения. Функция оценки `gbm_mcs_stat()` имеет единственный параметр: страйк-цена. В данном случае моделируется только уровень индекса на дату исполнения. В качестве ориентира выберем цену исполнения  $K = 105$ .

```
In [65]: S0 = 100.
```

```
        r = 0.05
```

```
        sigma = 0.25
```

```
        T = 1.0
```

```
        I = 50000
```

```
In [66]: def gbm_mcs_stat(K):
```

```
    ''' Оценка европейского колл-опциона в модели
        Блэка – Шоулза – Мертона по методу Монте-Карло
        (уровень индекса на дату исполнения)
```

```
    Параметры
```

```
    =====
```

```
    K: float
```

```
        (Положительная) страйк-цена опциона
```

```
    Возвращает
```

```
    =====
```

```
    C0: float
```

```
        Ожидаемая текущая стоимость европейского колл-опциона
```

```
    ...
```

```
    sn = gen_sn(1, I)
```

```
    # Моделирование уровня индекса на дату исполнения
```

```
    ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                    sigma * math.sqrt(T) * sn[1])
```

```
    # Вычисление выплаты на дату исполнения
```

```
    hT = np.maximum(ST - K, 0)
```

```
    # Вычисление оценки по методу Монте-Карло
```

```
    C0 = math.exp(-r * T) * np.mean(hT)
```

```
    return C0
```

```
In [67]: gbm_mcs_stat(K=105.) ❶
```

```
Out[67]: 10.044221852841922
```

❶ Оценка стоимости европейского колл-опциона по методу Монте-Карло.

Теперь можно переходить к динамическому моделированию процесса, включив в него не только европейский колл-опцион, но и пут-опцион. Соответствующий алгоритм реализован в функции `gbm_mcs_dyna()`. В следующем

коде также сравниваются прогнозируемые стоимости опционов “колл” и “пут” для одного и того же уровня индекса.

In [68]: M = 50 ❶

```
In [69]: def gbm_mcs_dyna(K, option='call'):
    ''' Оценка европейского колл-опциона в модели
        Блэка – Шоулза – Мертона по методу Монте-Карло
        (траектории уровней индекса)

    Параметры
    =====
    K: float
        (Положительная) страйк-цена опциона
    option: строка
        Тип оцениваемого опциона ('call', 'put')

    Возвращает
    =====
    C0: float
        Оценка текущей стоимости европейского колл-опциона
    '''
    dt = T / M
    # Моделирование траекторий уровня индекса
    S = np.zeros((M + 1, I))
    S[0] = S0
    sn = gen_sn(M, I)
    for t in range(1, M + 1):
        S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) *
                                   dt + sigma * math.sqrt(dt) * sn[t])
    # Расчет выплат по каждому типу
    if option == 'call':
        hT = np.maximum(S[-1] - K, 0)
    else:
        hT = np.maximum(K - S[-1], 0)
    # Вычисление оценки по методу Монте-Карло
    C0 = math.exp(-r * T) * np.mean(hT)
    return C0
```

In [70]: gbm\_mcs\_dyna(K=110., option='call') ❷

Out[70]: 7.950008525028434

```
In [71]: gbm_mcs_dyna(K=110., option='put') ❸  
Out[71]: 12.629934942682004
```

- ❶ Количество временных интервалов дискретизации.
- ❷ Оценка стоимости европейского колл-опциона по методу Монте-Карло.
- ❸ Оценка стоимости европейского пут-опциона по методу Монте-Карло.

Вопрос в том, насколько хорошо методы моделирования оценивают стоимость опционов в сравнении с эталонным значением, которое рассчитывается по формуле Блэка — Шоулза — Мертона? Чтобы выяснить это, в следующем коде мы сгенерируем соответствующие значения/оценки стоимости для диапазона страйк-цен, используя аналитическую функцию ценообразования европейских колл-опционов, содержащуюся в модуле `bsm_functions.py` (он рассматривается в конце главы).

Сначала сравним результаты статического моделирования с точными значениями, возвращаемыми аналитической функцией.

```
In [72]: from bsm_functions import bsm_call_value
```

```
In [73]: stat_res = [] ❶  
         dyna_res = [] ❶  
         anal_res = [] ❶  
         k_list = np.arange(80., 120.1, 5.) ❷  
         np.random.seed(100)
```

```
In [74]: for K in k_list:  
         stat_res.append(gbm_mcs_stat(K)) ❸  
         dyna_res.append(gbm_mcs_dyna(K)) ❸  
         anal_res.append(bsm_call_value(S0, K, T, r, sigma)) ❸
```

```
In [75]: stat_res = np.array(stat_res) ❹  
         dyna_res = np.array(dyna_res) ❹  
         anal_res = np.array(anal_res) ❹
```

- ❶ Инициализация пустых списков, предназначенных для хранения результатов.
- ❷ Создание массива `ndarray`, содержащего диапазон страйк-цен.
- ❸ Моделирование/вычисление значений стоимости опциона для всех страйк-цен.
- ❹ Преобразование списков в массивы `ndarray`.



Результаты представлены на рис. 12.15. Расхождения между прогнозируемыми и точными значениями составляют менее 1% как в положительную, так и в отрицательную сторону.

```
In [76]: plt.figure(figsize=(10, 6))
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
                               figsize=(10, 6))
ax1.plot(k_list, anal_res, 'b', label='Аналитические
                               вычисления')
ax1.plot(k_list, stat_res, 'ro', label='Статическое
                               моделирование')
ax1.set_ylabel('Стоимость европейского колл-опциона')
ax1.legend(loc=0)
ax1.set_ylim(bottom=0)
wi = 1.0
ax2.bar(k_list - wi / 2, (anal_res - stat_res) /
        anal_res * 100, wi)
ax2.set_xlabel('Цена исполнения')
ax2.set_ylabel('Разница, %')
ax2.set_xlim(left=75, right=125);

Out[76]: <Figure size 720x432 with 0 Axes>
```

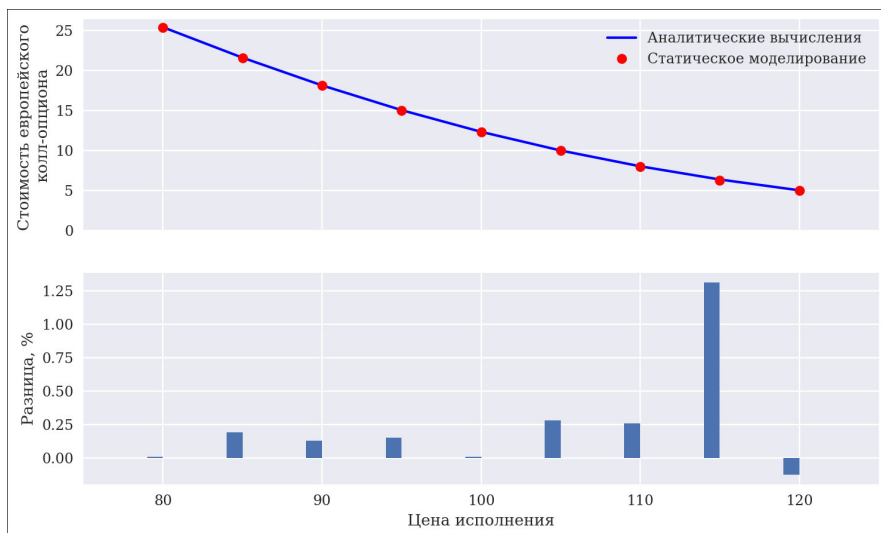
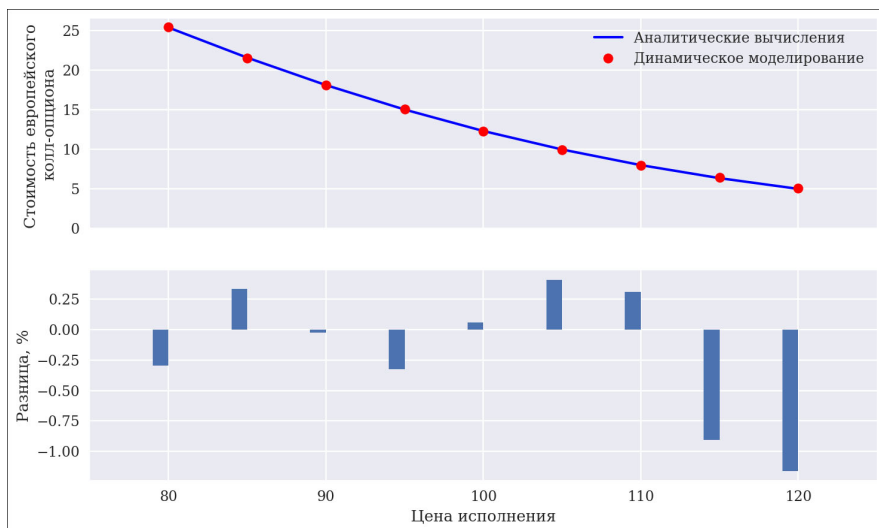


Рис. 12.15. Стоимость опциона, вычисляемая аналитическим способом и по методу Монте-Карло (статическое моделирование)

Похожая картина наблюдается и при сравнении результатов динамического моделирования с точными значениями стоимости, полученными аналитическим способом (рис. 12.16). Опять-таки, наблюдаемая разница не превышает 1% как в положительную, так и в отрицательную сторону. В общем случае точность оценки в методе Монте-Карло можно контролировать, меняя количество временных интервалов  $M$  и/или число траекторий  $I$ .

```
In [77]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
                                     figsize=(10, 6))
ax1.plot(k_list, anal_res, 'b', label='Аналитические
                                     вычисления')
ax1.plot(k_list, dyna_res, 'ro', label='Динамическое
                                     моделирование')
ax1.set_ylabel('Стоимость европейского колл-опциона')
ax1.legend(loc=0)
ax1.set_ylim(bottom=0)
wi = 1.0
ax2.bar(k_list - wi / 2, (anal_res - dyna_res) /
        anal_res * 100, wi)
ax2.set_xlabel('Цена исполнения')
ax2.set_ylabel('Разница, %')
ax2.set_xlim(left=75, right=125);
```



**Рис. 12.16.** Стоимость опциона, вычисляемая аналитическим способом и по методу Монте-Карло (динамическое моделирование)

## Американские опционы

Стоимость американского опциона рассчитывается немного сложнее по сравнению с европейским опционом. В данном случае для получения корректного значения необходимо определить *момент остановки* (марковский момент). Математически решение такой задачи описывается уравнением 12.12. Формулировка задачи основана на использовании дискретной временной сетки, участвующей в дальнейшем численном моделировании. В этом смысле корректнее говорить о *бермудском опционе*. Если временной интервал стремится к нулю, то стоимость бермудского опциона сходится к стоимости американского опциона.

**Уравнение 12.12. Ценообразование американского опциона как решение задачи оптимального момента остановки**

$$V_0 = \sup_{\tau \in \{0, \Delta t, 2\Delta t, \dots, T\}} e^{-rT} \mathbf{E}_0^Q \left( h_\tau (S_\tau) \right).$$

Приведенный ниже алгоритм основан на *методе наименьших квадратов Монте-Карло* (Least-Squares Monte Carlo — LSM) и взят из статьи Лонгстаффа и Шварца [4]. Можно показать, что стоимость американского (бермудского) опциона на любую заданную дату  $t$  вычисляется как  $V_t(s) = \max(h_t(s), C_t(s))$ , где  $C_t(s) = \mathbf{E}_t^Q \left( e^{-r\Delta t} V_{t+\Delta t}(S_{t+\Delta t}) \mid S_t = s \right)$  — *продленная стоимость* опциона при уровне индекса  $S_t = s$ .

Предположим, что мы смоделировали  $I$  траекторий уровня индекса на  $M$  временных отрезках равной длины  $\Delta t$ . Определим  $Y_{t,i} \equiv e^{-r\Delta t} V_{t+\Delta t,i}$  как моделируемую продленную стоимость опциона для траектории  $i$  в момент времени  $t$ . Мы не можем непосредственно задействовать данное значение, т.к. это означало бы идеальный прогноз. Но мы можем использовать перекрестные данные по всем моделируемым продленным стоимостям для оценки (ожидаемой) продленной стоимости путем регрессии по методу LSM.

Для заданного набора базисных функций  $b_d, d=1, \dots, D$  продленная стоимость вычисляется с помощью регрессионного уравнения  $\hat{C}_{t,i} = \sum_{d=1}^D \alpha_{d,t}^* \cdot b_d(S_{t,i})$ , где оптимальные регрессионные параметры  $\alpha^*$  являются решением уравнения 12.13.

**Уравнение 12.13. Регрессионная оценка американского опциона по методу наименьших квадратов**

$$\min_{\alpha_{1,t}, \dots, \alpha_{D,t}} \frac{1}{I} \sum_{i=1}^I \left( Y_{t,i} - \sum_{d=1}^D \alpha_{d,t} \cdot b_d(S_{t,i}) \right)^2.$$

Описанный алгоритм оценки американских опционов “колл” и “пут” методом наименьших квадратов (LSM) реализуется с помощью функции `gbm_mcs_amer()`.

```
In [78]: def gbm_mcs_amer(K, option='call'):
    ''' Оценка американского колл-опциона в модели
        Блэка – Шоулза – Мертона по методу
        наименьших квадратов Монте-Карло (LSM)

    Параметры
    =====
    K: float
        (Положительная) страйк-цена опциона
    option: строка
        Тип оцениваемого опциона ('call', 'put')

    Возвращает
    =====
    C0: float
        Оценка текущей стоимости американского колл-опциона
    '''
    dt = T / M
    df = math.exp(-r * dt)
    # Моделирование уровней индекса
    S = np.zeros((M + 1, I))
    S[0] = S0
    sn = gen_sn(M, I)
    for t in range(1, M + 1):
        S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) *
            . dt + sigma * math.sqrt(dt) * sn[t])
    # Расчет выплат по каждому типу
    if option == 'call':
        h = np.maximum(S - K, 0)
    else:
        h = np.maximum(K - S, 0)
    # Алгоритм LSM
    V = np.copy(h)
    for t in range(M - 1, 0, -1):
        reg = np.polyfit(S[t], V[t + 1] * df, 7)
        C = np.polyval(reg, S[t])
        V[t] = np.where(C > h[t], V[t + 1] * df, h[t])
    # Вычисление оценки по методу Монте-Карло
    C0 = df * np.mean(V[1])
    return C0
```

```
In [79]: gbm_mcs_amer(110., option='call')
Out[79]: 7.721705606305352
```

```
In [80]: gbm_mcs_amer(110., option='put')
Out[80]: 13.609997625418051
```

Стоимость европейского опциона представляет нижнюю границу стоимости американского опциона. Разница называется *премией досрочного исполнения*. В следующем коде сравниваются стоимости европейского и американского пут-опционов для прежнего диапазона страйк-цен, что позволяет оценить премию досрочного исполнения<sup>5</sup>.

```
In [81]: euro_res = []
         amer_res = []
```

```
In [82]: k_list = np.arange(80., 120.1, 5.)
```

```
In [83]: for K in k_list:
         euro_res.append(gbm_mcs_dyna(K, 'put'))
         amer_res.append(gbm_mcs_amer(K, 'put'))
```

```
In [84]: euro_res = np.array(euro_res)
         amer_res = np.array(amer_res)
```

Как показано на рис. 12.17, для рассматриваемого диапазона страйк-цен премия досрочного исполнения может достигать 10%.

```
In [85]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
         figsize=(10, 6))
         ax1.plot(k_list, euro_res, 'b',
                 label='Европейский пут-опцион')
         ax1.plot(k_list, amer_res, 'go',
                 label='Американский пут-опцион')
         ax1.set_ylabel('Стоимость пут-опциона')
         ax1.legend(loc=0)
         wi = 1.0
         ax2.bar(k_list - wi / 2, (amer_res - euro_res) /
                 euro_res * 100, wi)
         ax2.set_xlabel('Цена исполнения')
         ax2.set_ylabel('Премия досрочного исполнения, %')
         ax2.set_xlim(left=75, right=125);
```

---

<sup>5</sup> Поскольку выплаты дивидендов не предусмотрены, для колл-опционов нет премии досрочного исполнения (нет причин досрочно исполнять опцион).

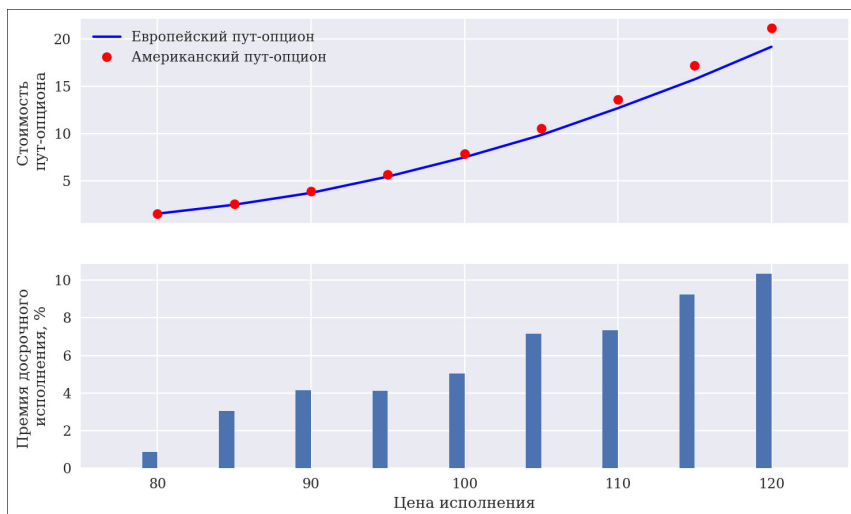


Рис. 12.17. Сравнение американского и европейского опционов

## Оценка рисков

Не менее важной областью применения стохастических методов моделирования, наряду с оценкой опционов, является *управление рисками*. В этом разделе мы познакомимся со способами вычисления/оценки двух наиболее распространенных в финансовом анализе мер риска.

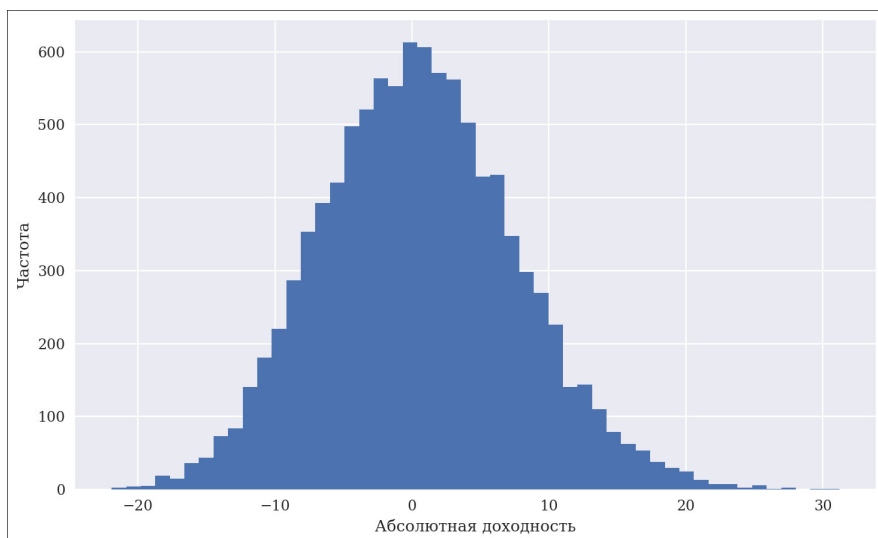
### Стоимость под риском

*Стоимость под риском* (Value at Risk — VaR) — одна из самых популярных и одновременно противоречивых мер риска. Аналитики-практики любят ее за интуитивную понятность, тогда как теоретики активно критикуют ее за невозможность оценить размер убытков вне доверительного уровня (*хвост распределения*). Таким образом, VaR — это числовое значение, выраженное в денежных единицах (например, USD, EUR, JPY) и характеризующее потери (портфеля, отдельной позиции и т.п.), которые с заданным уровнем доверия (вероятности) не будут превышены в течение определенного периода времени.

Рассмотрим биржевую позицию величиной 1 млн долл., для которой значение VaR составляет 50 тыс. долл. с уровнем доверия 99% на отрезке 30 дней (один месяц). Это означает, что в течение 30 дней ожидаемые потери *не превысят* 50 тыс. долларов с вероятностью 99% (т.е. в 99 случаях из 100). В то же

время ничего не известно о том, какими будут убытки, если потери превысят 50 тыс. долларов. Например, какова вероятность того, что потери составят 100 или 500 тыс. долларов? Единственное, о чем говорит VaR, — что убытки, *превышающие* 50 тыс. долларов, возникают с вероятностью 1%.

Применим модель Блэка — Шоулза — Мертона для оценки уровней индекса на будущий период  $T = 30 / 365$  (30 дней). Чтобы оценить VaR, необходимо смоделировать абсолютные значения прибылей и убытков относительно текущей позиции, представив их в отсортированном виде, т.е. от самого большого убытка до самой крупной прибыли. На рис. 12.18 показана полученная гистограмма.



*Рис. 12.18. Смоделированные абсолютные значения прибылей и убытков (геометрическое броуновское движение)*

```
In [86]: S0 = 100
         r = 0.05
         sigma = 0.25
         T = 30 / 365.
         I = 10000
```

```
In [87]: ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
         sigma * np.sqrt(T) * npr.standard_normal(I)) ❶
```

```
In [88]: R_gbm = np.sort(ST - S0) ❷
```

```
In [89]: plt.figure(figsize=(10, 6))
plt.hist(R_gbm, bins=50)
plt.xlabel('Абсолютная доходность')
plt.ylabel('Частота');
```

- ❶ Значения на конец периода, полученные для модели геометрического броуновского движения.
- ❷ Вычисление абсолютных значений прибылей и убытков по каждой траектории и сортировка результатов.

Располагая массивом `ndarray`, содержащим отсортированные результаты моделирования, дальнейшие вычисления можно выполнить с помощью функции `scs.scoreatpercentile()`. Нужно лишь определить соответствующие проценти. В следующем коде проценти хранятся в списке `percs`. Например, значение `0.1` соответствует уровню доверия  $100\% - 0,1\% = 99,9\%$ . В данном случае на 30-дневном отрезке значение VaR для уровня доверия  $99,9\%$  равно `18.8`, а для уровня  $90\%$  — `8.5`.

```
In [91]: percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
var = scs.scoreatpercentile(R_gbm, percs)
print('%21s %11s' % ('Доверительный уровень', 'VaR'))
print(33 * '-')
for pair in zip(percs, var):
    print('%21.2f %11.3f' % (100 - pair[0], -pair[1]))
Доверительный уровень      VaR
-----
          99.99          21.814
          99.90          18.837
          99.00          15.230
          97.50          12.816
          95.00          10.824
          90.00           8.504
```

Во втором примере, в котором мы возвращаемся к прыжковой диффузии Мертона, моделирование выполняется динамическим способом. В данном случае, из-за того что подверженный скачкам компонент имеет отрицательное среднее значение, моделируемые значения прибылей и убытков демонстрируют бимодальное распределение (рис. 12.19). С точки зрения нормального распределения здесь наблюдается *толстый левый хвост*.

```
In [92]: dt = 30. / 365 / M
rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1)
```

```
In [93]: S = np.zeros((M + 1, I))
```



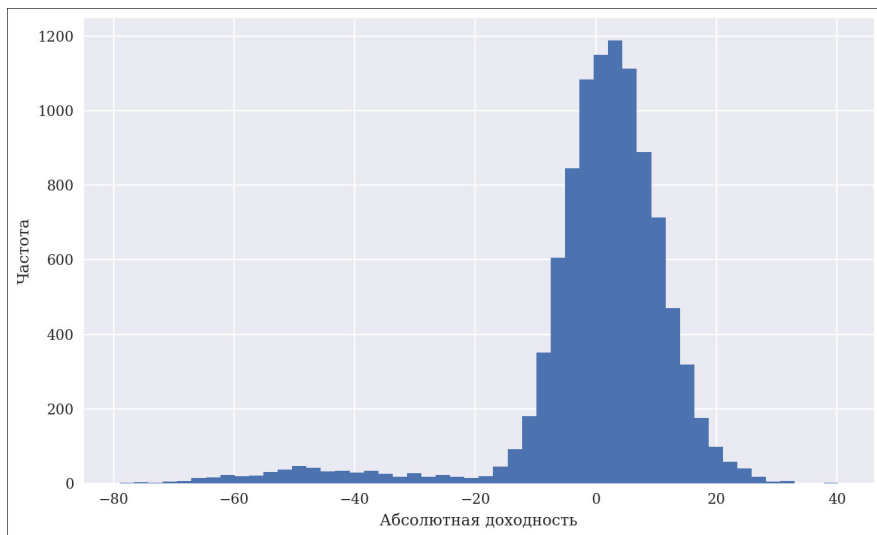
```

S[0] = S0
sn1 = npr.standard_normal((M + 1, I))
sn2 = npr.standard_normal((M + 1, I))
poi = npr.poisson(lamb * dt, (M + 1, I))
for t in range(1, M + 1, 1):
    S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * \
        sigma ** 2) * dt + sigma * math.sqrt(dt) * \
        sn1[t]) + (np.exp(mu + delta * sn2[t]) - 1) * \
        poi[t])
    S[t] = np.maximum(S[t], 0)

```

```
In [94]: R_jd = np.sort(S[-1] - S0)
```

```
In [95]: plt.figure(figsize=(10, 6))
plt.hist(R_jd, bins=50)
plt.xlabel('Абсолютная доходность')
plt.ylabel('Частота');
```



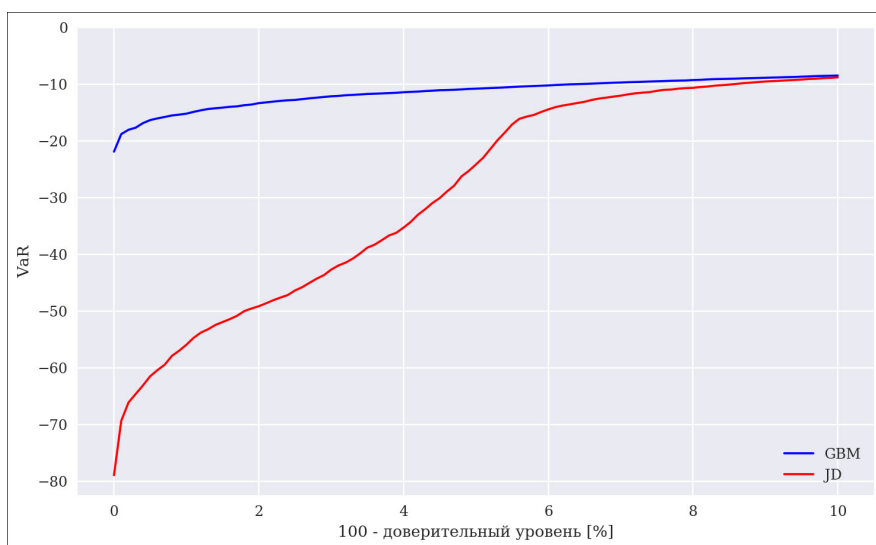
**Рис. 12.19.** Смоделированные абсолютные значения прибылей и убытков (прыжковая диффузия)

При такой параметризации процесса значение VaR, определяемое на 30-дневном отрезке для уровня доверия 90%, получается почти таким же, как и в случае геометрического броуновского движения, а вот при уровне доверия 99,9% оно оказывается более чем *в три раза* выше (70 против 18.8).

```
In [96]: percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
var = scs.scoreatpercentile(R_jd, percs)
print('%21s %11s' % ('Доверительный уровень', 'VaR'))
print(33 * '-')
for pair in zip(percs, var):
    print('%21.2f %11.3f' % (100 - pair[0], -pair[1]))
Доверительный уровень      VaR
-----
                        99.99      76.520
                        99.90      69.396
                        99.00      55.974
                        97.50      46.405
                        95.00      24.198
                        90.00       8.836
```

Это хорошо иллюстрирует проблему оценки хвостовых рисков, часто возникающую при работе со стандартной мерой VaR.

Дополнительной иллюстрацией служит рис. 12.20, на котором сравниваются две меры VaR. Легко заметить, что они ведут себя совершенно по-разному в одном и том же диапазоне доверительных уровней.



**Рис. 12.20.** Стоимость под риском, полученная для геометрического броуновского движения (GBM) и прыжковой диффузии (JD)

```
In [97]: percs = list(np.arange(0.0, 10.1, 0.1))
        gbm_var = scs.scoreatpercentile(R_gbm, percs)
        jd_var = scs.scoreatpercentile(R_jd, percs)

In [98]: plt.figure(figsize=(10, 6))
        plt.plot(percs, gbm_var, 'b', lw=1.5, label='GBM')
        plt.plot(percs, jd_var, 'r', lw=1.5, label='JD')
        plt.legend(loc=4)
        plt.xlabel('(100 - доверительный уровень), %')
        plt.ylabel('VaR')
        plt.ylim(ymax=0.0);
```

## Поправка на кредитный риск

Другими важными мерами риска служат *кредитная стоимость под риском* (Credit Value-at-Risk — CVaR) и *поправка на кредитный риск* (Credit Valuation Adjustment — CVA), которая выводится из CVaR. Грубо говоря, CVaR характеризует риск невыполнения контрагентом своих обязательств, например в результате банкротства. В таком случае делаются два основных предположения: *вероятность дефолта* и *(средний) уровень потерь*.

Рассмотрим конкретный пример, снова обратившись к модели Блэка — Шоулза — Мертона. В простейшем случае принимается фиксированный (средний) уровень потерь  $L$  и фиксированная вероятность  $p$  наступления дефолта контрагента (за год). Такой сценарий реализован в следующем коде, в котором используется распределение Пуассона и учитывается тот факт, что дефолт может наступить лишь раз.

```
In [99]: S0 = 100.
        r = 0.05
        sigma = 0.2
        T = 1.
        I = 100000

In [100]: ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * \
        np.sqrt(T) * npr.standard_normal(I))

In [101]: L = 0.5 ❶

In [102]: p = 0.01 ❷

In [103]: D = npr.poisson(p * T, I) ❸

In [104]: D = np.where(D > 1, 1, D) ❹
```

- ❶ Определение уровня потерь.
- ❷ Определение вероятности дефолта.
- ❸ Моделирование дефолтных ситуаций.
- ❹ Ограничение дефолтных ситуаций единственным случаем.

В отсутствие дефолта риск-нейтральное значение будущего уровня индекса должно быть равно текущему значению актива (не считая различий, связанных с погрешностью вычислений). Значение CVaR и текущая стоимость актива с поправкой на кредитный риск вычисляются следующим образом.

```
In [105]: math.exp(-r * T) * np.mean(ST) ❶
Out[105]: 99.94767178982691
```

```
In [106]: CVaR = math.exp(-r * T) * np.mean(L * D * ST) ❷
          CVaR ❷
Out[106]: 0.4883560258963962
```

```
In [107]: S0_CVA = math.exp(-r * T) * np.mean((1 - L * D) * ST) ❸
          S0_CVA ❸
Out[107]: 99.45931576393053
```

```
In [108]: S0_adj = S0 - CVaR ❹
          S0_adj ❹
Out[108]: 99.5116439741036
```

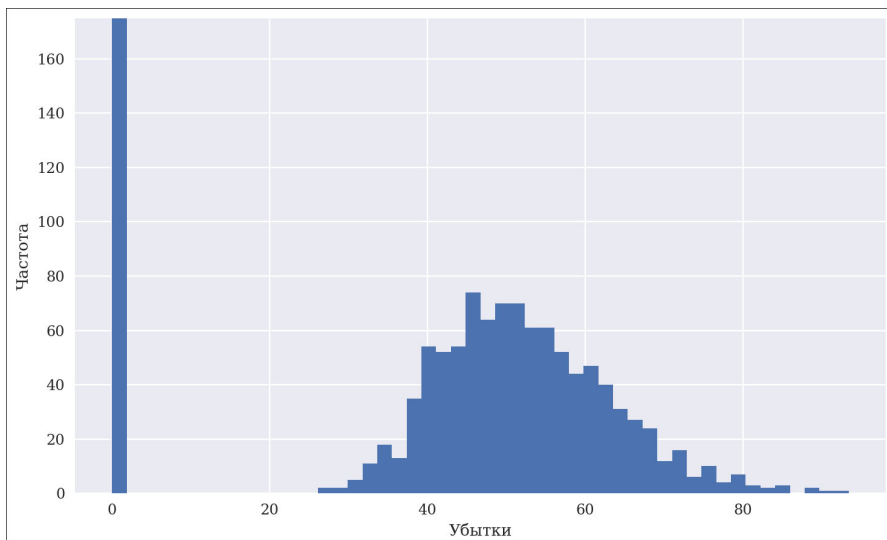
- ❶ Дисконтированное среднее смоделированное значение актива в момент времени  $T$ .
- ❷ Значение CVaR как дисконтированное среднее будущих потерь в случае дефолта.
- ❸ Дисконтированное среднее смоделированное значение актива в момент времени  $T$ , скорректированное на смоделированные потери от дефолта.
- ❹ Текущая цена актива, скорректированная смоделированным значением CVaR.

В данном примере потери, связанные с кредитными рисками, наблюдаются приблизительно в 1000 случаях, что и следовало ожидать, учитывая предполагаемую вероятность дефолта 1% и 100 000 моделируемых траекторий. Полное частотное распределение потерь, вызванных дефолтом, приведено на рис. 12.21. Конечно, в подавляющем большинстве случаев (примерно 99 000 из 100 000) никаких потерь не происходит.

```
In [109]: np.count_nonzero(L * D * ST) ❶
Out[109]: 978
```

```
In [110]: plt.figure(figsize=(10, 6))
plt.hist(L * D * ST, bins=50)
plt.xlabel('Убытки')
plt.ylabel('Частота')
plt.ylim(ymax=175);
```

❶ Количество дефолтных событий и связанных с ними потерь.



*Рис. 12.21. Убытки, вызванные риск-нейтральным ожидаемым дефолтом (индекса)*

Теперь рассмотрим случай европейского колл-опциона. Его стоимость равна 10.4 при страйк-цене 100. Тогда показатель CVaR будет составлять 0.05 при тех же самых предположениях, касающихся вероятности наступления дефолта и уровне потерь.

```
In [111]: K = 100.
hT = np.maximum(ST - K, 0)
```

```
In [112]: C0 = math.exp(-r * T) * np.mean(hT) ❶
C0 ❶
```

```
Out[112]: 10.396916492839354
```

```
In [113]: CVaR = math.exp(-r * T) * np.mean(L * D * hT) ❷
```

```
CVaR ❷
```

```
Out[113]: 0.05159099858923533
```

```
In [114]: C0_CVA = math.exp(-r * T) * np.mean((1 - L * D) * hT) ❸
```

```
C0_CVA ❸
```

```
Out[114]: 10.34532549425012
```

- ❶ Оценка стоимости европейского колл-опциона, полученная методом Монте-Карло.
- ❷ Значение CVaR как дисконтированное среднее будущих потерь в случае дефолта.
- ❸ Оценка стоимости европейского колл-опциона, полученная методом Монте-Карло и скорректированная с учетом возможных потерь от дефолта.

По сравнению с обычным активом опцион проявляет несколько иные характеристики. Потери, вызванные дефолтом, наблюдаются примерно в 500 случаях, несмотря на то что всего насчитывается 1000 дефолтных событий. Это обусловлено высокой вероятностью нулевых выплат по опциону при его экспирации. Как показано на рис. 12.22, показатель CVaR для опциона имеет совершенно иное частотное распределение, чем в случае обычного актива.

```
In [115]: np.count_nonzero(L * D * hT) ❶
```

```
Out[115]: 538
```

```
In [116]: np.count_nonzero(D) ❷
```

```
Out[116]: 978
```

```
In [117]: I - np.count_nonzero(hT) ❸
```

```
Out[117]: 44123
```

```
In [118]: plt.figure(figsize=(10, 6))  
plt.hist(L * D * hT, bins=50)  
plt.xlabel('Убытки')  
plt.ylabel('Частота')  
plt.ylim(ymax=350);
```

- ❶ Количество потерь, вызванных дефолтом.
- ❷ Количество дефолтных событий.
- ❸ Количество случаев безрезультатой экспирации опциона (с нулевыми выплатами).

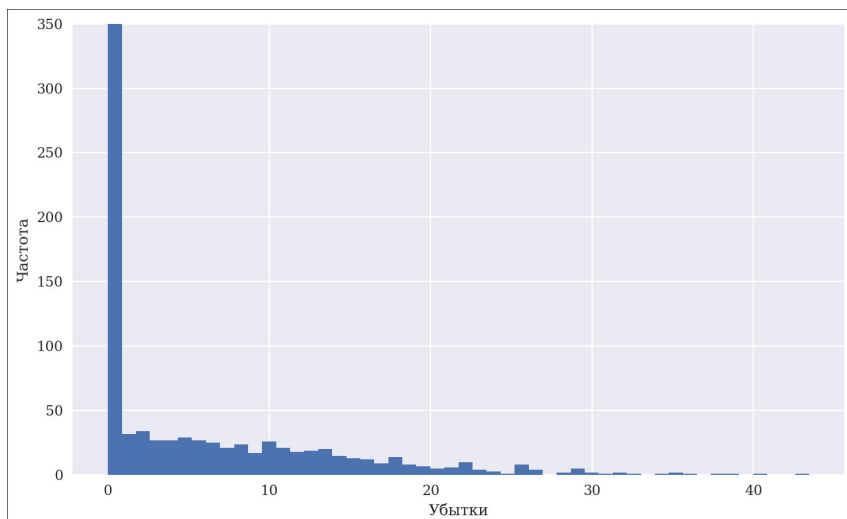


Рис. 12.22. Убытки, вызванные риск-нейтральным ожидаемым дефолтом (колл-опциона)

## Общий сценарий Python

Ниже показана реализация основных аналитических функций, связанных с ценообразованием европейских колл-опционов в модели Блэка — Шоулза — Мертона. Подробное описание модели приведено в работах Блэка и Шоулза [1], а также Мертона [5]. Альтернативный способ реализации на основе класса Python приведен в приложении Б.

```
#
# Оценка европейского колл-опциона в модели
# Блэка — Шоулза — Мертона (BSM), включая функцию
# вычисления веги и функцию прогнозирования
# ожидаемой волатильности
#
# bsm_functions.py
#
# (c) Dr. Yves J. Hilpisch
# Python for Finance, 2nd ed.
#
```

```

def bsm_call_value(S0, K, T, r, sigma):
    ''' Оценка европейского колл-опциона в модели
        Блэка – Шоулза – Мертона, аналитическая формула.

    Параметры
    =====
    S0: float
        Начальный уровень акции/индекса
    K: float
        Страйк-цена
    T: float
        Срок исполнения (в долях года)
    r: float
        Постоянная безрисковая краткосрочная ставка
    sigma: float
        Коэффициент волатильности компонента диффузии

    Возвращает
    =====
    value: float
        Текущая стоимость европейского колл-опциона
    '''

    from math import log, sqrt, exp
    from scipy import stats

    S0 = float(S0)
    d1 = (log(S0 / K) + (r + 0.5 * sigma ** 2) * T) /
        (sigma * sqrt(T))
    d2 = (log(S0 / K) + (r - 0.5 * sigma ** 2) * T) /
        (sigma * sqrt(T))

    # stats.norm.cdf -> функция распределения для
    # нормального распределения

    value = (S0 * stats.norm.cdf(d1, 0.0, 1.0) -
             K * exp(-r * T) * stats.norm.cdf(d2, 0.0, 1.0))
    return value

def bsm_vega(S0, K, T, r, sigma):
    ''' Вега европейского колл-опциона в модели
        Блэка – Шоулза – Мертона.

```



*Параметры*

=====

*S0: float*

*Начальный уровень акции/индекса*

*K: float*

*Страйк-цена*

*T: float*

*Срок исполнения (в долях года)*

*r: float*

*Постоянная безрисковая краткосрочная ставка*

*sigma: float*

*Коэффициент волатильности компонента диффузии*

*Возвращает*

=====

*vega: float*

*Частная производная формулы BSM по коэффициенту сигма  
(т.е. веха)*

'''

```
from math import log, sqrt
```

```
from scipy import stats
```

```
S0 = float(S0)
```

```
d1 = (log(S0 / K) + (r + 0.5 * sigma ** 2) * T) /  
      (sigma * sqrt(T))
```

```
vega = S0 * stats.norm.pdf(d1, 0.0, 1.0) * sqrt(T)
```

```
return vega
```

*# Функция вычисления ожидаемой волатильности*

```
def bsm_call_imp_vol(S0, K, T, r, C0, sigma_est, it=100):
```

```
''' Ожидаемая волатильность европейского колл-опциона  
    в модели Блэка – Шоулза – Мертона.
```

*Параметры*

=====

*S0: float*

*Начальный уровень акции/индекса*

*K: float*

*Страйк-цена*

*T: float*

*Срок исполнения (в долях года)*

*r: float*

*Постоянная безрисковая краткосрочная ставка*

```

sigma_est: float
    Прогнозируемая ожидаемая волатильность
it: int
    Число итераций

Возвращает
=====
sigma_est: float
    Численный прогноз ожидаемой волатильности
'''
for i in range(it):
    sigma_est -= ((bsm_call_value(S0, K, T, r, sigma_est) - C0) /
                  bsm_vega(S0, K, T, r, sigma_est))
return sigma_est

```

## Резюме

В этой главе рассказывалось о применении метода Монте-Карло в финансовом анализе. Сначала было показано, как генерировать псевдослучайные числа на основе различных законов распределения. Далее рассматривались способы моделирования случайных величин и стохастических процессов, что находит широкое применение во многих финансовых задачах. Мы углубленно изучили две прикладные области: оценка европейских и американских опционов и оценка мер риска, таких как стоимость под риском (VaR) и поправка на кредитный риск (CVA).

Мы убедились в том, что Python в сочетании с библиотекой NumPy прекрасно подходит для реализации даже таких вычислительно сложных задач, как оценка американских опционов по методу Монте-Карло. Это в основном связано с тем, что большинство функций и классов NumPy реализовано на C, за счет чего можно добиться намного более высокой производительности по сравнению с чистым кодом Python. Дополнительным преимуществом становится компактность и понятность кода благодаря векторизованным операциям.

## Дополнительные ресурсы

Исходная статья, в которой рассматривается применение метода Монте-Карло в финансовом моделировании:

- Boyle, Phelim. “Options: A Monte Carlo Approach” (1977, *Journal of Financial Economics*, Vol. 4, No. 3, pp. 323–338).

В главе упоминались следующие источники.

1. Black, Fischer, and Myron Scholes. “The Pricing of Options and Corporate Liabilities” (1973, *The Journal of Political Economy*, Vol. 81, No. 3, pp. 637–654).
2. Cox, John, Jonathan Ingersoll, and Stephen Ross. “A Theory of the Term Structure of Interest Rates” (1985, *Econometrica*, Vol. 53, No. 2, pp. 385–407).
3. Heston, Steven. “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options” (1993, *The Review of Financial Studies*, Vol. 6, No. 2, pp. 327–343).
4. Longstaff, Francis, and Eduardo Schwartz. “Valuing American Options by Simulation: A Simple Least Squares Approach” (2001, *Review of Financial Studies*, Vol. 14, No. 1, pp. 113–147).
5. Merton, Robert. “Theory of Rational Option Pricing” (1973, *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141–183).
6. Merton, Robert. “Option Pricing When the Underlying Stock Returns Are Discontinuous” (1976, *Journal of Financial Economics*, Vol. 3, No. 3, pp. 125–144).

В следующих книгах более подробно раскрыты темы, рассмотренные в этой главе (в первой книге не рассматриваются технические детали реализации).

- Glasserman, Paul. *Monte Carlo Methods in Financial Engineering* (2004, Springer).
- Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).

Исчерпывающее описание кредитных рисков приведено в следующей книге:

- Duffie, Darrell, and Kenneth Singleton. *Credit Risk: Pricing, Measurement, and Management* (2003, Princeton University Press).

---

## Статистический анализ

С помощью статистики я могу доказать что угодно, кроме истины.

*Джордж Каннинг*

Статистический анализ — это обширная область знаний. Соответствующие инструменты и получаемые с их помощью результаты оказываются незаменимыми при работе с финансовыми данными. Это объясняет популярность таких предметно-ориентированных языков программирования, как R. Чем сложнее становятся применяемые статистические модели, тем выше потребность в простых и производительных решениях.

Даже целая глава не позволит в полной мере раскрыть все многообразие данной области математики. Поэтому, как и во многих других главах, мы сконцентрируемся только на самых важных темах и рассмотрим применение Python для решения конкретных финансовых задач. Ключевых тем будет четыре.

### *Нормальное распределение*

Большинство популярных финансовых моделей, таких как портфельная теория Марковица и модель ценообразования капитальных активов, основывается на предположении о нормальном распределении значений доходности ценных бумаг. Следовательно, необходимо иметь возможность проверить, соответствует ли временной ряд нормальному распределению.

### *Оптимизация портфеля*

Портфельная теория Марковица считается одним из самых больших достижений в финансовой статистике. Разработанная в начале 1950-х годов пионером отрасли Гарри Марковицем, эта теория была призвана противопоставить точные математические расчеты и статистические методы анализа распространенной на тот момент практике принятия самостоятельных инвестиционных решений. В определенном смысле она стала первой настоящей количественной моделью, получившей широкое распространение в финансовой отрасли.

## Байесовская статистика

На концептуальном уровне байесовская теория вводит понятие *степени доверия* к случайному событию, которая может меняться при получении новых статистических данных. Применительно к линейной регрессии это может означать наличие статистического распределения для регрессионных параметров (таких, как угловой коэффициент и коэффициент сдвига регрессионной прямой) вместо оценочных значений отдельных точек. На сегодняшний день байесовские методы находят широкое применение в финансах, и в этом разделе мы рассмотрим несколько примеров.

## Машинное обучение

Машинное обучение базируется на передовых статистических методах и относится к классу методов искусственного интеллекта (ИИ). Подобно математической статистике машинное обучение предлагает обширный набор методик исследования наборов данных и построения прогнозов на основе полученных результатов. Алгоритмы машинного обучения могут относиться к разным направлениям, таким как *обучение с учителем* или *обучение без учителя*. Типы задач, решаемых с помощью таких алгоритмов, тоже различаются. Самые популярные задачи — *кластеризация* и *классификация*. Мы будем рассматривать *алгоритмы классификации на основе обучения с учителем*.

В этой главе мы будем обрабатывать значения даты/времени. Соответствующие инструменты Python, включая библиотеки NumPy и pandas, описаны в приложении А.

# Нормальное распределение

*Нормальное распределение* играет важнейшую роль в финансовых вычислениях, это одна из основ финансовой теории. На предположении о том, что доходность финансового инструмента подчиняется нормальному распределению<sup>1</sup>, основаны следующие финансовые модели.

## Портфельная теория Марковица

В случае нормального распределения доходностей акций для принятия инвестиционных решений (касательно оптимального состава портфеля)

---

<sup>1</sup> Другая ключевая гипотеза — *линейность*. Например, предполагается, что финансовые рынки, как правило, демонстрируют линейную зависимость между спросом на акции и уплачиваемой за них ценой. Другими словами, в большинстве случаев рынки считаются полностью ликвидными в том смысле, что меняющийся спрос не оказывает никакого влияния на удельную стоимость финансового инструмента.

достаточно ориентироваться только на (ожидаемую) *среднюю доходность, дисперсию* (волатильность) доходностей и *ковариацию* между доходностями различных акций.

### *Модель ценообразования капитальных активов*

Опять-таки, при нормальном распределении доходностей акций цены отдельных позиций можно поставить в линейную зависимость от фондового индекса. Такая зависимость обычно выражается с помощью *бета-коэффициента* (характеризует зависимость между доходностью отдельной ценной бумаги и доходностью всего рыночного портфеля).

### *Гипотеза эффективного рынка*

*Эффективным* считается рынок, на котором курсовые стоимости ценных бумаг отражают всю доступную информацию, при этом понятие “вся доступная информация” можно трактоваться более узко или более широко (например, “вся общедоступная информация” или “вся инсайдерская информация”). Если гипотеза справедлива, то цены на акции колеблются случайным образом, а распределение доходностей оказывается нормальным.

### *Модель ценообразования опционов Блэка — Шоулза*

Броуновское движение — *эталонная модель*, применяемая для изучения флуктуаций цен финансовых инструментов. В общеизвестной формуле оценки опционов Блэка — Шоулза — Мертона модель геометрического броуновского движения используется для описания флуктуаций цен во времени, что ведет к логнормальному распределению цен и нормальному распределению доходностей.

Даже этот далеко не полный список позволяет оценить важность предположения о нормальном распределении данных.

## **Эталонный портфель**

Чтобы подготовить почву для дальнейшего анализа, начнем с исследования геометрического броуновского движения как одного из хрестоматийных стохастических процессов, применяемых в финансовом моделировании. Для траекторий процессов, моделируемых геометрическим броуновским движением  $S$ , справедливы следующие утверждения.

### *Нормальное распределение логарифмических доходностей*

Логарифмические доходности  $\log \frac{S_t}{S_s} = \log S_t - \log S_s$  для периода времени  $0 < s < t$  подчинены *нормальному* распределению.

### Логнормальное распределение стоимостей

Для любого значения  $t > 0$  стоимости  $S_t$  подчиняются логнормальному распределению.

Прежде чем выполнять примеры, необходимо задать настройки вывода и импортировать ряд пакетов, включая `scipy.stats` (<http://docs.scipy.org/doc/scipy/reference/stats.html>) и `statsmodels.api` (<http://statsmodels.sourceforge.net/stable/>).

```
In [1]: import math
import numpy as np
import scipy.stats as scs
import statsmodels.api as sm
from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

В следующем коде моделируемые траектории геометрического броуновского движения (см. главу 12), рассчитываемые по методу Монте-Карло, генерируются с помощью функции `gen_paths()`.

```
In [3]: def gen_paths(S0, r, sigma, T, M, I):
''' Генерирование траекторий геометрического
броуновского движения по методу Монте-Карло

Параметры
=====
S0: float
    Начальный уровень акции/индекса
r: float
    Постоянная краткосрочная ставка
sigma: float
    Постоянная волатильность
T: float
    Конечный временной горизонт
M: int
    Количество временных шагов/интервалов
I: int
    Количество моделируемых траекторий
```

```

Возвращает
=====
paths: ndarray, shape (M + 1, I)
    Траектории, смоделированные согласно заданным параметрам
'''
dt = T / M
paths = np.zeros((M + 1, I))
paths[0] = S0
for t in range(1, M + 1):
    rand = np.random.standard_normal(I)
    rand = (rand - rand.mean()) / rand.std() ❶
    paths[t] = paths[t - 1] * np.exp((r - 0.5 * sigma **
        2) * dt + sigma * math.sqrt(dt) * rand) ❷
return paths

```

- ❶ Сопоставление первого и второго моментов.
- ❷ Векторизованная дискретизация геометрического броуновского движения по методу Эйлера.

Процесс моделирования основан на параметризации модели по методу Монте-Карло, как показано ниже. Функция `gen_paths()` генерирует 250 000 траекторий, каждая из которых состоит из 50 временных шагов. Графики первых десяти траекторий приведены на рис. 13.1.



**Рис. 13.1.** Десять первых траекторий геометрического броуновского движения



```
In [4]: S0 = 100. ①
        r = 0.05 ②
        sigma = 0.2 ③
        T = 1.0 ④
        M = 50 ⑤
        I = 250000 ⑥
        np.random.seed(1000)
```

```
In [5]: paths = gen_paths(S0, r, sigma, T, M, I)
```

```
In [6]: S0 * math.exp(r * T) ⑦
Out[6]: 105.12710963760242
```

```
In [7]: paths[-1].mean() ⑦
Out[7]: 105.12645392478755
```

```
In [8]: plt.figure(figsize=(10, 6))
        plt.plot(paths[:, :10])
        plt.xlabel('Время')
        plt.ylabel('Индекс');
```

- ① Начальное значение для моделируемых процессов.
- ② Постоянная краткосрочная ставка.
- ③ Постоянный коэффициент волатильности.
- ④ Временной горизонт (интервал) в долях года.
- ⑤ Количество временных шагов.
- ⑥ Количество моделируемых процессов.
- ⑦ Ожидаемое значение и среднее смоделированное значение.

Главный интерес для нас будет представлять распределение логарифмических доходностей. На данный момент данные о логарифмической доходности, полученные при моделировании траекторий, хранятся в объекте `ndarray`. Ниже приведены значения логарифмической доходности для отдельной траектории.

```
In [9]: paths[:, 0].round(4)
Out[9]: array([100.    ,  97.821 ,  98.5573, 106.1546, 105.899 ,
              99.8363,
              100.0145, 102.6589, 105.6643, 107.1107, 108.7943,
```

```

108.2449,
106.4105, 101.0575, 102.0197, 102.6052, 109.6419,
109.5725,
112.9766, 113.0225, 112.5476, 114.5585, 109.942 ,
112.6271,
112.7502, 116.3453, 115.0443, 113.9586, 115.8831,
117.3705,
117.9185, 110.5539, 109.9687, 104.9957, 108.0679,
105.7822,
105.1585, 104.3304, 108.4387, 105.5963, 108.866 ,
108.3284,
107.0077, 106.0034, 104.3964, 101.0637, 98.3776,
97.135 ,
95.4254, 96.4271, 96.3386])

```

```
In [10]: log_returns = np.log(paths[1:] / paths[:-1])
```

```
In [11]: log_returns[:, 0].round(4)
```

```

Out[11]: array([-0.022 ,  0.0075,  0.0743, -0.0024, -0.059 ,  0.0018,
                0.0261,
                0.0289,  0.0136,  0.0156, -0.0051, -0.0171, -0.0516,
                0.0095,
                0.0057,  0.0663, -0.0006,  0.0306,  0.0004, -0.0042,
                0.0177,
                -0.0411,  0.0241,  0.0011,  0.0314, -0.0112, -0.0095,
                0.0167,
                0.0128,  0.0047, -0.0645, -0.0053, -0.0463,  0.0288,
                -0.0214,
                -0.0059, -0.0079,  0.0386, -0.0266,  0.0305, -0.0049,
                -0.0123,
                -0.0094, -0.0153, -0.0324, -0.0269, -0.0127, -0.0178,
                0.0104,
                -0.0009])

```

Такой результат вполне можно получить на реальных финансовых рынках: в одни дни инвестиции в акции приносят *прибыль*, а в другие вы терпите *убытки* от предыдущих вложений.

Функция `print_statistics()` — это оболочка для функции `scs.describe()` из пакета `scipy.stats`. Она генерирует наглядную статистическую сводку для заданного набора данных (исторических или смоделированных), включающую такие показатели, как среднее, коэффициент асимметрии, коэффициент эксцесса и др.

```
In [13]: def print_statistics(array):
''' Вывод основных статистических показателей.

Параметры
=====
array: ndarray
    Объект, для которого генерируются
    статистические показатели
'''

sta = scs.describe(array)
print('%14s %15s' % ('Статистика', 'Значение'))
print(30 * '-')
print('%14s %15.5f' % ('size', sta[0]))
print('%14s %15.5f' % ('min', sta[1][0]))
print('%14s %15.5f' % ('max', sta[1][1]))
print('%14s %15.5f' % ('mean', sta[2]))
print('%14s %15.5f' % ('std', np.sqrt(sta[3])))
print('%14s %15.5f' % ('skew', sta[4]))
print('%14s %15.5f' % ('kurtosis', sta[5]))
```

```
In [14]: print_statistics(log_returns.flatten())
```

Статистика	Значение
size	12500000.00000
min	-0.15664
max	0.15371
mean	0.00060
std	0.02828
skew	0.00055
kurtosis	0.00085

```
In [15]: log_returns.mean() * M + 0.5 * sigma ** 2 ❶
Out[15]: 0.050000000000000005
```

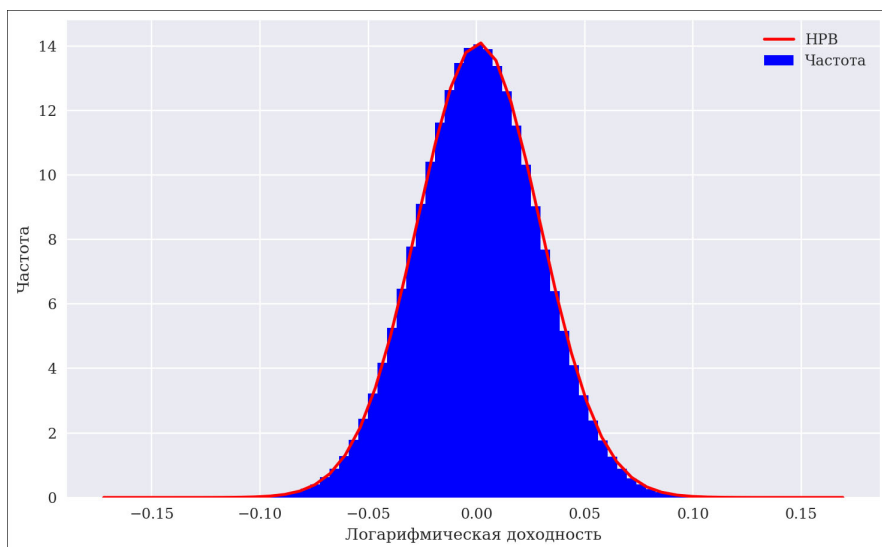
```
In [16]: log_returns.std() * math.sqrt(M) ❷
Out[16]: 0.200000000000000015
```

- ❶ Среднегодовая логарифмическая доходность после коррекции по формуле Ито<sup>2</sup>.
- ❷ Годовая волатильность, т.е. годовое стандартное отклонение логарифмической доходности.

<sup>2</sup> Основы стохастического исчисления Ито описаны у Глассермана [6].

В рассмотренном выше случае набор данных состоит из 12,5 млн точек, представленных преимущественно значениями из диапазона от  $-0,15$  до  $+0,15$ . Для него можно ожидать среднегодовые значения средней доходности на уровне  $0,05$  (после коррекции по формуле Ито) и стандартного отклонения (волатильности) на уровне  $0,2$ . В итоге годовые значения почти полностью соответствуют этому (умножьте среднее значение на 50 и скорректируйте его по формуле Ито, а стандартное отклонение умножьте на  $\sqrt{50}$ ). Одна из причин настолько хорошего соответствия заключается в согласовании моментов при понижении дисперсии в процессе генерирования случайных чисел (см. главу 12).

На рис. 13.2 приведено сравнение частотного распределения смоделированных значений логарифмической доходности с плотностью нормального распределения вероятностей (НРВ) для заданных значений параметров  $\gamma$  и  $\sigma$ . Все необходимые вычисления выполняются функцией `poisson.pdf()` из пакета `scipy.stats`. Легко заметить, что совпадение распределений достаточно точное.



**Рис. 13.2.** Гистограмма логарифмической доходности в модели геометрического броуновского движения и график плотности нормального распределения вероятностей

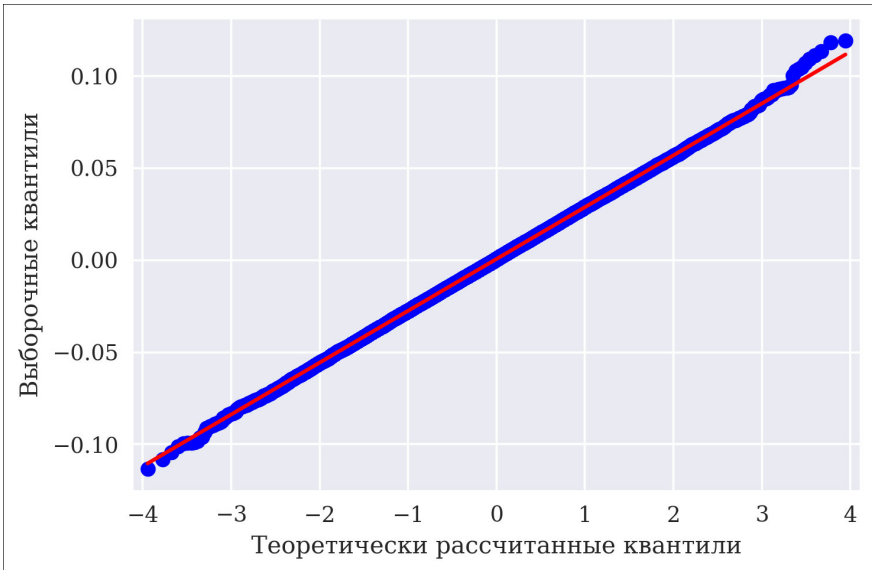
```
In [17]: plt.figure(figsize=(10, 6))
plt.hist(log_returns.flatten(), bins=70, normed=True,
        label='Частота', color='b')
plt.xlabel('Логарифмическая доходность')
```

```
plt.ylabel('Частота')
x = np.linspace(plt.axis()[0], plt.axis()[1])
plt.plot(x, scs.norm.pdf(x, loc=r / M, scale=sigma /
    np.sqrt(M)), 'r', lw=2.0, label='НРБ') ❶
plt.legend();
```

- ❶ Построение графика плотности нормального распределения вероятностей для предполагаемых параметров, масштабированных на длину интервала.

Сравнение частотного распределения (гистограммы) с теоретической плотностью вероятности — это не единственный графический способ выявления нормального распределения. Другое полезное средство — так называемые графики “квантиль — квантиль” (Q-Q). В следующем примере смоделированные квантили сравниваются с теоретически рассчитанными. Для набора значений с нормальным распределением подавляющее большинство точек будет лежать в непосредственной близости к теоретическому графику, представленному прямой линией (рис. 13.3).

```
In [18]: sm.qqplot(log_returns.flatten()[:500], line='s')
plt.xlabel('Теоретически рассчитанные квантили')
plt.ylabel('Выборочные квантили');
```



*Рис. 13.3. График “квантиль — квантиль” для логарифмической доходности в модели геометрического броуновского движения*

Как бы ни были привлекательны графические способы анализа, они, как правило, не могут стать заменой более строгим процедурам тестирования. Применяемая в следующем примере функция `normality_tests()` выполняет сразу три статистических теста.

*Проверка коэффициента асимметрии (`skewtest()`)*

В этом тесте проверяется “нормальность” коэффициента асимметрии для выборочных данных (т.е. близок ли он к нулю).

*Проверка коэффициента эксцесса (`kurtosistest()`)*

В этом тесте проверяется коэффициент эксцесса для выборочных данных (опять-таки, близок ли он к нулю).

*Проверка нормального распределения (`normaltest()`)*

Объединяет две предыдущие проверки.

Полученные в результате проверки Р-значения свидетельствуют о том, что логарифмическая доходность, описываемая геометрическим броуновским движением, действительно подчиняется нормальному распределению (все критерии больше 0,05).

```
In [19]: def normality_tests(arr):  
        ''' Проверка нормального распределения для набора данных.
```

```
        Параметры
```

```
        =====
```

```
        array: ndarray
```

```
        Объект, для которого генерируются  
        статистические показатели
```

```
        ...
```

```
        print('Коэффициент асимметрии набора данных %14.3f'  
              % scs.skew(arr))
```

```
        print('Р-значение коэффициента асимметрии %14.3f' %  
              scs.skewtest(arr)[1])
```

```
        print('Коэффициент эксцесса набора данных %14.3f'  
              % scs.kurtosis(arr))
```

```
        print('Р-значение коэффициента эксцесса %14.3f' %  
              scs.kurtosistest(arr)[1])
```

```
        print('Р-значение критерия нормальности %14.3f' %  
              scs.normaltest(arr)[1])
```

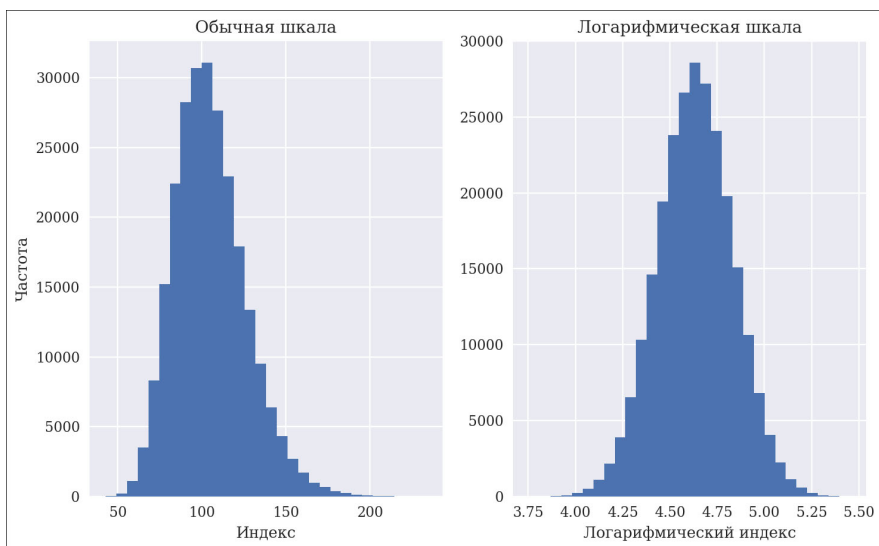
```
In [20]: normality_tests(log_returns.flatten()) ❶  
Коэффициент асимметрии набора данных      0.001  
Р-значение коэффициента асимметрии          0.430
```

Коэффициент эксцесса набора данных	0.001
P-значение коэффициента эксцесса	0.541
P-значение критерия нормальности	0.607

❶ Все P-значения намного больше 0,05.

Наконец, необходимо проверить, подчиняются ли значения на конец периода логнормальному распределению. Такая проверка сводится к определению критерия нормальности: достаточно прологарифмировать данные и убедиться в том, что полученные значения имеют (или не имеют) нормальное распределение. На рис. 13.4 показаны гистограммы уровней индекса на конец периода, как подчиняющиеся логнормальному распределению, так и прологарифмированные напрямую.

```
In [21]: f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))
ax1.hist(paths[-1], bins=30)
ax1.set_xlabel('Индекс')
ax1.set_ylabel('Частота')
ax1.set_title('Обычная шкала')
ax2.hist(np.log(paths[-1]), bins=30)
ax2.set_xlabel('Логарифмический индекс')
ax2.set_title('Логарифмическая шкала')
```



*Рис. 13.4. Гистограммы уровней индекса на конец периода, полученные путем моделирования геометрического броуновского движения*

Статистические показатели исходного набора данных демонстрируют вполне ожидаемое поведение, например среднее значение близко к 105. Значения логарифмического индекса имеют коэффициенты асимметрии и эксцесса, близкие к нулю, и демонстрируют высокие Р-значения, что подтверждает гипотезу о нормальном распределении.

```
In [22]: print_statistics(paths[-1])
```

Статистика	Значение
size	250000.00000
min	42.74870
max	233.58435
mean	105.12645
std	21.23174
skew	0.61116
kurtosis	0.65182

```
In [23]: print_statistics(np.log(paths[-1]))
```

Статистика	Значение
size	250000.00000
min	3.75534
max	5.45354
mean	4.63517
std	0.19998
skew	-0.00092
kurtosis	-0.00327

```
In [24]: normality_tests(np.log(paths[-1]))
```

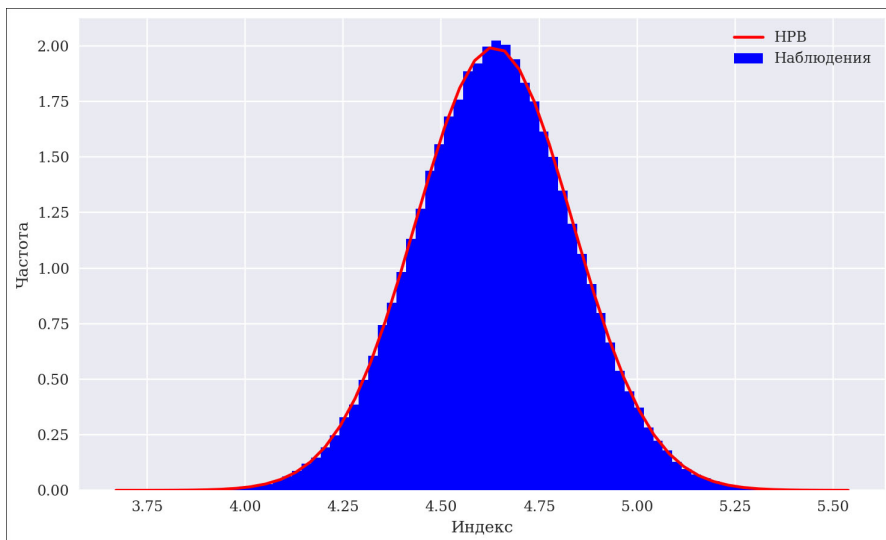
Коэффициент асимметрии набора данных	-0.001
Р-значение коэффициента асимметрии	0.851
Коэффициент эксцесса набора данных	-0.003
Р-значение коэффициента эксцесса	0.744
Р-значение критерия нормальности	0.931

На рис. 13.5 показано, что частотное распределение достаточно точно совпадает с плотностью нормального распределения вероятностей (как и предполагалось).

```
In [25]: plt.figure(figsize=(10, 6))
log_data = np.log(paths[-1])
plt.hist(log_data, bins=70, normed=True,
         label='Наблюдения', color='b')
plt.xlabel('Индекс')
```



```
plt.ylabel('Частота')
x = np.linspace(plt.axis()[0], plt.axis()[1])
plt.plot(x, scs.norm.pdf(x, log_data.mean(),
                        log_data.std()), 'r', lw=2.0, label='HPB')
plt.legend();
```



*Рис. 13.5. Гистограмма уровней логарифмического индекса в модели геометрического броуновского движения и график плотности нормального распределения вероятностей*

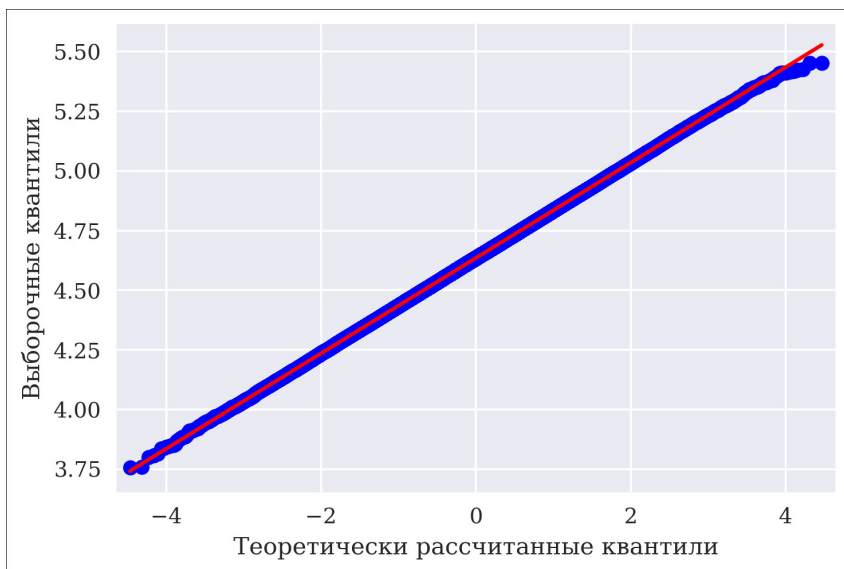
На рис. 13.6 также можно увидеть доказательство гипотезы о нормальном распределении уровней логарифмического индекса.

```
In [26]: sm.qqplot(log_data, line='s')
plt.xlabel('Теоретически рассчитанные квантили')
plt.ylabel('Выборочные квантили');
```



### Гипотеза о нормальном распределении

На предположении о нормальном распределении неизвестных значений доходности финансовых инструментов строится целый ряд финансовых теорий. Python располагает эффективными статистическими и графическими средствами проверки того, подчиняются ли данные временных рядов нормальному распределению.



*Рис. 13.6. График “квантиль — квантиль” для уровней логарифмического индекса, моделируемых геометрическим броуновским движением*

## Существующие исторические данные

В этом разделе мы проанализируем четыре финансовых временных ряда. Два из них содержат котировки акций технологических компаний, а другие два — котировки биржевых инвестиционных фондов:

- APPL.0 — Apple, Inc.;
- MSFT.0 — Microsoft, Inc.;
- SPY — SPDR S&P 500 Trust;
- GLD — SPDR Gold Trust.

Для управления данными обратимся к программным инструментам библиотеки `pandas` (см. главу 8). На рис. 13.7 показано, как изменяются нормализованные цены на финансовые инструменты во времени.

```
In [27]: import pandas as pd
```

```
In [28]: raw = pd.read_csv('../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True).dropna()
```

```
In [29]: symbols = ['SPY', 'GLD', 'AAPL.O', 'MSFT.O']
```

```
In [30]: data = raw[symbols]
        data = data.dropna()
```

```
In [31]: data.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
Data columns (total 4 columns):
SPY          2138 non-null float64
GLD          2138 non-null float64
AAPL.O       2138 non-null float64
MSFT.O       2138 non-null float64
dtypes: float64(4)
memory usage: 83.5 KB
```

```
In [32]: data.head()
```

```
Out[32]:
```

	SPY	GLD	AAPL.O	MSFT.O
Date				
2010-01-04	113.33	109.80	30.572827	30.950
2010-01-05	113.63	109.70	30.625684	30.960
2010-01-06	113.71	111.51	30.138541	30.770
2010-01-07	114.19	110.82	30.082827	30.452
2010-01-08	114.57	111.37	30.282827	30.660

```
In [33]: (data / data.iloc[0] * 100).plot(figsize=(10, 6))
```

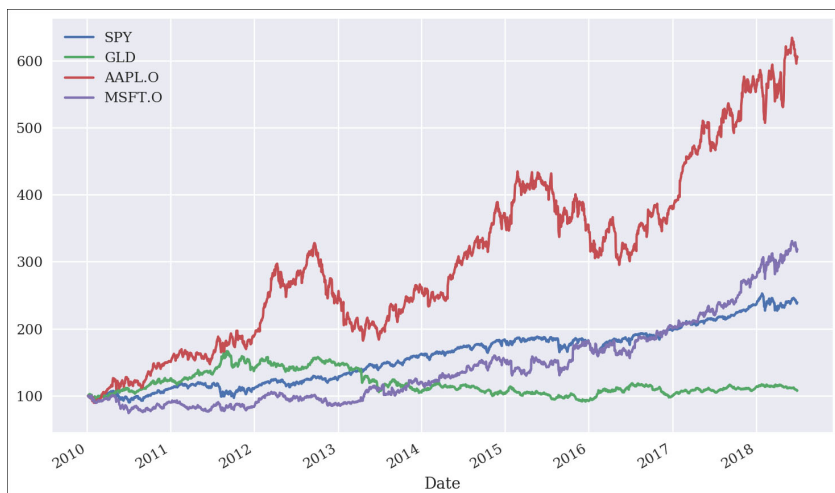
На рис. 13.8 приведены гистограммы изменения логарифмической доходности финансовых инструментов во времени.

```
In [34]: log_returns = np.log(data / data.shift(1))
        log_returns.head()
```

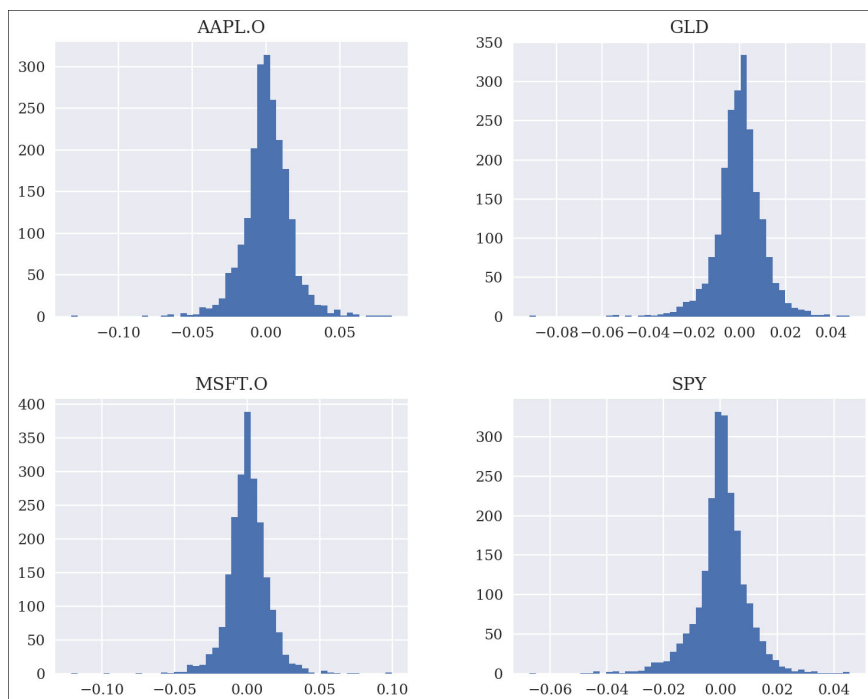
```
Out[34]:
```

	SPY	GLD	AAPL.O	MSFT.O
Date				
2010-01-04	NaN	NaN	NaN	NaN
2010-01-05	0.002644	-0.000911	0.001727	0.000323
2010-01-06	0.000704	0.016365	-0.016034	-0.006156
2010-01-07	0.004212	-0.006207	-0.001850	-0.010389
2010-01-08	0.003322	0.004951	0.006626	0.006807

```
In [35]: log_returns.hist(bins=50, figsize=(10, 8));
```



*Рис. 13.7. Изменение нормализованных цен на финансовые инструменты во времени*



*Рис. 13.8. Гистограммы логарифмической доходности финансовых инструментов*

Теперь рассмотрим различные статистические показатели временных рядов. Заметьте, что во всех четырех случаях значение коэффициента эксцесса не характерно для нормального распределения.

```
In [36]: for sym in symbols:
          print('\nРезультаты для символа {}'.format(sym))
          print(30 * '-')
          log_data = np.array(log_returns[sym].dropna())
          print_statistics(log_data) ❶
```

Результаты для символа SPY

-----	
Статистика	Значение
-----	
size	2137.00000
min	-0.06734
max	0.04545
mean	0.00041
std	0.00933
skew	-0.52189
kurtosis	4.52432

Результаты для символа GLD

-----	
Статистика	Значение
-----	
size	2137.00000
min	-0.09191
max	0.04795
mean	0.00004
std	0.01020
skew	-0.59934
kurtosis	5.68423

Результаты для символа AAPL.O

-----	
Статистика	Значение
-----	
size	2137.00000
min	-0.13187
max	0.08502
mean	0.00084
std	0.01591

skew	-0.23510
kurtosis	4.78964

Результаты для символа MSFT.O

Статистика	Значение
size	2137.00000
min	-0.12103
max	0.09941
mean	0.00054
std	0.01421
skew	-0.09117
kurtosis	7.29106

# ❶ Статистические показатели временных рядов финансовых инструментов.

На рис. 13.9 приведен график “квантиль — квантиль” (Q-Q) для тикера SPY. Вполне очевидно, что выборочные квантили не лежат на прямой линии, а это указывает на то, что исходные данные не подчиняются нормальному распределению. Точки в левой и правой частях графика отдаляются от прямой в область соответственно меньших и больших значений. Другими словами, временные ряды характеризуются *утяжеленными хвостами*. Этот термин пришел из статистики, где он обозначает ту часть распределения, в которой наблюдаются сильные расхождения (в отрицательную или положительную сторону) с нормальным распределением. То же самое можно сказать и о графике для акций Microsoft (рис. 13.10). На нем также наблюдаются утяжеленные хвосты.

```
In [37]: sm.qqplot(log_returns['SPY'].dropna(), line='s')
plt.title('SPY')
plt.xlabel('Теоретически рассчитанные квантили')
plt.ylabel('Выборочные квантили');
```

```
In [38]: sm.qqplot(log_returns['MSFT.O'].dropna(), line='s')
plt.title('MSFT.O')
plt.xlabel('Теоретически рассчитанные квантили')
plt.ylabel('Выборочные квантили');
```

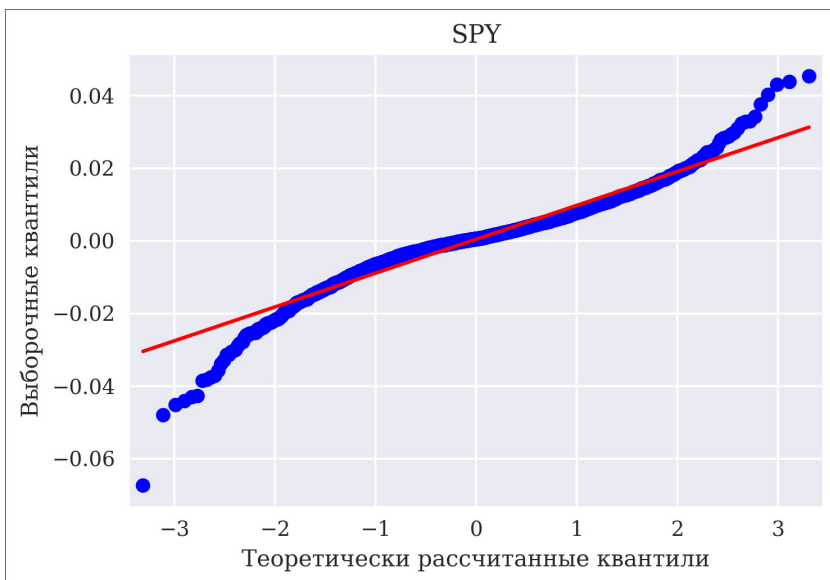


Рис. 13.9. График “квантиль — квантиль” для логарифмической доходности акций SPY

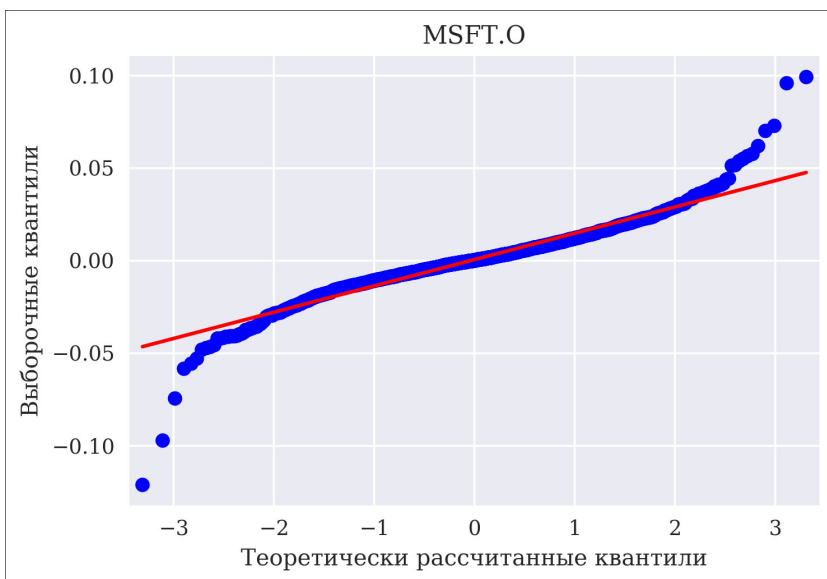


Рис. 13.10. График “квантиль — квантиль” для логарифмической доходности акций MSFT.O

Наконец, можно выполнить статистические тесты.

```
In [39]: for sym in symbols:
          print('\nРезультаты для символа {}'.format(sym))
          print(32 * '-')
          log_data = np.array(log_returns[sym].dropna())
          normality_tests(log_data) ❶
```

Результаты для символа SPY

```
-----
Кoeffициент асимметрии набора данных      -0.522
P-значение коэффицента асимметрии          0.000
Кoeffициент эксцесса набора данных         4.524
P-значение коэффицента эксцесса            0.000
P-значение критерия нормальности           0.000
```

Результаты для символа GLD

```
-----
Кoeffициент асимметрии набора данных      -0.599
P-значение коэффицента асимметрии          0.000
Кoeffициент эксцесса набора данных         5.684
P-значение коэффицента эксцесса            0.000
P-значение критерия нормальности           0.000
```

Результаты для символа AAPL.0

```
-----
Кoeffициент асимметрии набора данных      -0.235
P-значение коэффицента асимметрии          0.000
Кoeffициент эксцесса набора данных         4.790
P-значение коэффицента эксцесса            0.000
P-значение критерия нормальности           0.000
```

Результаты для символа MSFT.0

```
-----
Кoeffициент асимметрии набора данных      -0.091
P-значение коэффицента асимметрии          0.085
Кoeffициент эксцесса набора данных         7.291
P-значение коэффицента эксцесса            0.000
P-значение критерия нормальности           0.000
```

- ❶ Проверка P-значений для временных рядов нескольких финансовых инструментов.



R-значения во всех тестах равны нулю, а значит, гипотезу о нормальном распределении временных рядов следует *решиительно отвергнуть*. Это еще раз показывает, что предположение о нормальном распределении доходностей акций и других классов активов (как, например, в модели геометрического броуновского движения) нельзя принимать безоговорочно. Во многих случаях приходится обращаться к более сложным моделям (в частности, моделям прыжковой диффузии или моделям со стохастической волатильностью), способным генерировать утяжеленные хвосты.

## Оптимизация портфеля

Портфельная теория Марковица — краеугольный камень финансового анализа. В 1990 году Гарри Марковиц получил Нобелевскую премию по экономике за ее теоретическое обоснование. Несмотря на то что портфельная теория была сформулирована еще в 1950-х годах, ее до сих пор преподают студентам финансовых специальностей, и она находит широкое практическое применение (обычно с теми или иными модификациями)<sup>3</sup>. В этом разделе мы рассмотрим основные положения теории.

Введение в портфельную теорию изложено в главе 5 книги Коупленда, Уэстона и Шастри [3]. Как указывалось ранее, ключевым постулатом теории является предположение о нормальном распределении значений доходности.

Рассматривая только среднее значение и дисперсию, мы неизбежно предполагаем, что для математического описания распределения капитала на конец периода не понадобятся никакие другие статистические показатели. Если только инвесторы не располагают специальной функцией полезности (а именно квадратичной), следует предполагать, что доходность имеет нормальное распределение, которое можно целиком описать с помощью среднего и дисперсии.

## Данные

В последующих примерах будут использоваться те же финансовые инструменты, что и ранее. Основная идея портфельной теории Марковица заключается в *диверсификации* с целью достижения минимального портфельного риска при заданном уровне целевой доходности или максимальной доходности портфеля при заданном уровне риска. Можно ожидать, что положительный эффект от такой диверсификации будет достигаться при большом числе разнообразных активов. Но в нашем случае для демонстрации ключевых

---

<sup>3</sup> См. работу Марковица [9].

принципов будет достаточно четырех финансовых инструментов. Частотное распределение их логарифмических доходностей показано на рис. 13.11.

```
In [40]: symbols = ['AAPL.O', 'MSFT.O', 'SPY', 'GLD'] ❶
```

```
In [41]: noa = len(symbols) ❷
```

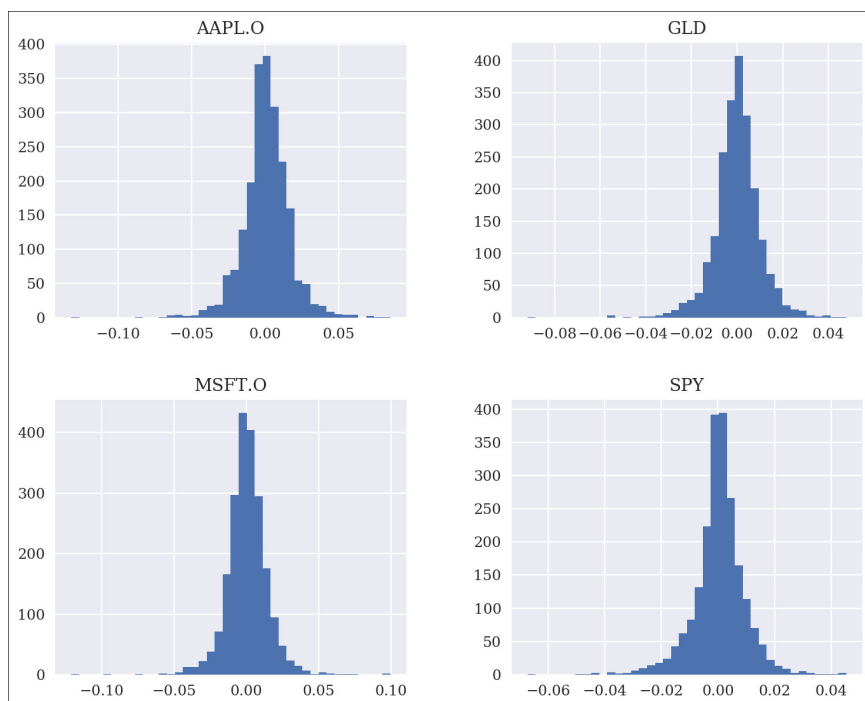
```
In [42]: data = raw[symbols]
```

```
In [43]: rets = np.log(data / data.shift(1))
```

```
In [44]: rets.hist(bins=40, figsize=(10, 8));
```

❶ Четыре финансовых инструмента в структуре портфеля.

❷ Количество финансовых инструментов.



*Рис. 13.11. Гистограммы логарифмической доходности выбранных финансовых инструментов*

Процесс выбора портфеля основывается на изучении *ковариационной матрицы* инвестиционных финансовых инструментов. Библиотека `pandas` располагает встроенным методом генерирования ковариационной матрицы, к которой можно применить тот же самый коэффициент масштабирования.

```
In [45]: rets.mean() * 252 ❶
```

```
Out[45]: AAPL.O    0.212359
         MSFT.O    0.136648
         SPY      0.102928
         GLD      0.009141
         dtype: float64
```

```
In [46]: rets.cov() * 252 ❷
```

```
Out[46]:
```

	AAPL.O	MSFT.O	SPY	GLD
AAPL.O	0.063773	0.023427	0.021039	0.001513
MSFT.O	0.023427	0.050917	0.022244	-0.000347
SPY	0.021039	0.022244	0.021939	0.000062
GLD	0.001513	-0.000347	0.000062	0.026209

❶ Среднегодовая доходность.

❷ Годовая ковариационная матрица.

## Теоретическое обоснование

В дальнейшем предполагается, что инвестору не разрешается открывать короткие позиции по финансовому инструменту. Разрешены только длинные позиции, а это значит, 100% инвестиционного капитала нужно распределить между доступными инструментами так, чтобы все позиции были длинными (положительными) и в сумме составляли 100%. Располагая четырьмя инструментами, можно, например, инвестировать в них равные суммы: по 25% доступного капитала на каждый. В следующем фрагменте генерируются четыре случайных числа из равномерного распределения в диапазоне от 0 до 1, после чего значения нормализуются так, чтобы их сумма равнялась 1.

```
In [47]: weights = np.random.random(noa) ❶
```

```
         weights /= np.sum(weights) ❷
```

```
In [48]: weights
```

```
Out[48]: array([0.07650728, 0.06021919, 0.63364218, 0.22963135])
```

```
In [49]: weights.sum()
```

```
Out[49]: 1.0
```

- ❶ Случайные веса инструментов в портфеле...
- ❷ ...нормализуемые в диапазоне от 1 до 100%.

Как указывалось выше, сумма весов должна равняться 1, т.е.  $\sum_i^I w_i = 1$ , где  $I$  — количество финансовых инструментов, а  $w_i > 0$  — вес финансового инструмента  $i$ . В уравнении 13.1 приведена формула *ожидаемой доходности портфеля*, учитывающая веса всех финансовых инструментов. Доходность считается ожидаемой в том смысле, что историческая средняя доходность здесь рассматривается как наилучшая оценка будущей доходности. В этой формуле  $r_i$  — зависящие от состояния портфеля будущие доходности (вектор значений доходности, для которых предполагается нормальное распределение), а  $\mu_i$  — ожидаемая доходность инструмента  $i$ . Наконец,  $w^T$  — это транспонированный вектор весов, а  $\mu$  — вектор ожидаемых доходностей ценных бумаг.

**Уравнение 13.1. Общая формула ожидаемой доходности портфеля**

$$\begin{aligned}\mu_p &= E\left(\sum_i w_i r_i\right) = \\ &= \sum_i w_i E(r_i) = \\ &= \sum_i w_i \mu_i = \\ &= w^T \mu.\end{aligned}$$

В переводе на Python такая формула представляется единственной инструкцией (включающей пересчет в годовом выражении).

```
In [50]: np.sum(rets.mean() * weights) * 252 ❶
Out[50]: 0.09179459482057793
```

- ❶ Годовая доходность портфеля с учетом весов финансовых инструментов.

Другой важный элемент портфельной теории Марковица — *дисперсия ожидаемой доходности портфеля*. Если ковариация между двумя ценными бумагами определяется как  $\sigma_{ij} = \sigma_{ji} = E((r_i - \mu_i)(r_j - \mu_j))$ , то дисперсию отдельной ценной бумаги можно представить как ковариацию с ней самой, поэтому  $\sigma_i^2 = E((r_i - \mu_i)^2)$ . Вид ковариационной матрицы для портфеля ценных бумаг (при условии, что доли ценных бумаг одинаковы и равны 1) показан в уравнении 13.2.

### Уравнение 13.2. Ковариационная матрица портфеля

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1I} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{I1} & \sigma_{I2} & & \sigma_I^2 \end{bmatrix}$$

Подставив ковариационную матрицу в исходное уравнение, можно получить конечную формулу вычисления дисперсии ожидаемой доходности (уравнение 13.3).

### Уравнение 13.3. Общая формула вычисления дисперсии ожидаемой доходности портфеля

$$\begin{aligned} \sigma_p^2 &= \mathbb{E}\left((r - \mu)^2\right) = \\ &= \sum_{i \in I} \sum_{j \in I} w_i w_j \sigma_{ij} = \\ &= w^T \Sigma w. \end{aligned}$$

В Python такого рода формула тоже представляется одной строкой кода за счет векторизованных средств NumPy. В частности, функция `np.dot()` возвращает скалярное произведение двух векторов/матриц. Транспонирование вектора или матрицы осуществляется с помощью метода `transpose()` или атрибута `T`. После вычисления дисперсии портфеля остается извлечь квадратный корень  $\sigma_p = \sqrt{\sigma_p^2}$  для получения (ожидаемого) стандартного отклонения, или волатильности, портфеля.

```
In [51]: np.dot(weights.T, np.dot(rets.cov() * 252, weights)) ❶  
Out[51]: 0.014763288666485574
```

```
In [52]: math.sqrt(np.dot(weights.T, np.dot(rets.cov() * 252,  
                                             weights))) ❷  
Out[52]: 0.12150427427249452
```

- ❶ Годовая дисперсия портфеля при заданных весах финансовых инструментов.
- ❷ Годовая волатильность портфеля при заданных весах финансовых инструментов.



## Python и векторизация вычислений

Рассмотренный выше пример наглядно показывает, насколько легко транслировать такие математические понятия, как доходность или дисперсия портфеля, в векторизованный код Python (см. главу 1).

На этом завершается рассмотрение инструментов, применяемых для статистических расчетов в рамках портфельной теории Марковица. Инвесторов в первую очередь интересует, какие соотношения риска и доходности можно получить для заданного набора финансовых инструментов и с какими статистическими показателями. Ответ можно получить, применив метод Монте-Карло (см. главу 12) для полномасштабного генерирования векторов случайных весов финансовых инструментов. В следующем коде ожидаемая доходность и дисперсия фиксируются отдельно для каждого моделируемого профиля. Чтобы упростить код, мы определим две вспомогательные функции: `port_ret()` и `port_vol()`.

```
In [53]: def port_ret(weights):  
         return np.sum(rets.mean() * weights) * 252  
  
In [54]: def port_vol(weights):  
         return np.sqrt(np.dot(weights.T, np.dot(rets.cov() *  
         252, weights)))  
  
In [55]: prets = []  
         pvols = []  
         for p in range(2500): ❶  
             weights = np.random.random(noa) ❶  
             weights /= np.sum(weights) ❶  
             prets.append(port_ret(weights)) ❷  
             pvols.append(port_vol(weights)) ❷  
         prets = np.array(prets)  
         pvols = np.array(pvols)
```

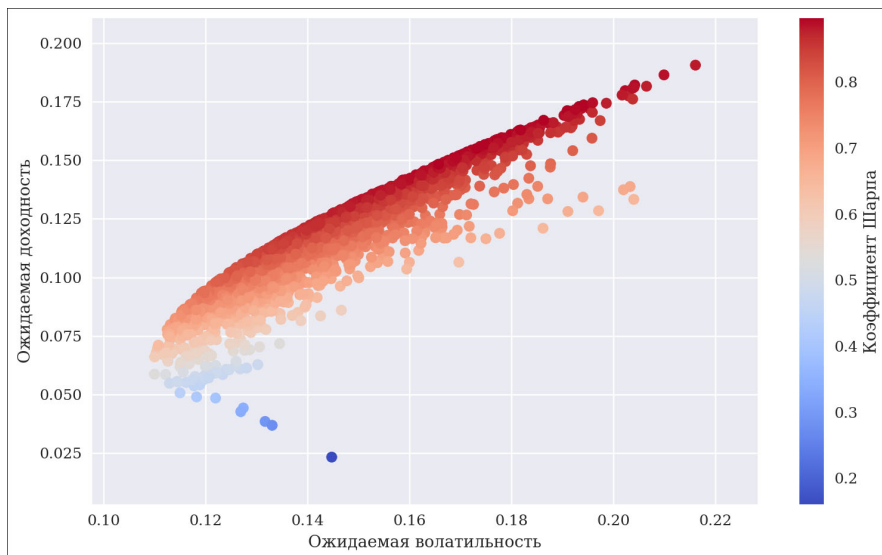
- ❶ Моделирование весов портфеля по методу Монте-Карло.
- ❷ Сохранение статистических показателей в списках.

Результаты моделирования приведены на рис. 13.12. Рядом с диаграммой построена также шкала коэффициента Шарпа, который определяется как отношение премии за риск (ожидаемое превышение доходности портфеля над безрисковой краткосрочной ставкой  $r_f$ ) к ожидаемому стандартному

отклонению и вычисляется по формуле  $SR \equiv \frac{\mu_p - r_f}{\sigma_p}$ . Для простоты будем считать, что  $r_f \equiv 0$ .

```
In [56]: plt.figure(figsize=(10, 6))
plt.scatter(pvols, pretss, c=pretss / pvols,
            marker='o', cmap='coolwarm')

plt.xlabel('Ожидаемая волатильность')
plt.ylabel('Ожидаемая доходность')
plt.colorbar(label='Коэффициент Шарпа');
```



**Рис. 13.12.** Ожидаемые доходности и волатильности портфелей со случайными весами финансовых инструментов

Из рис. 13.12 становится ясно, что не всякое распределение весов дает хорошие результаты. Например, для уровня риска 15% можно получить большое количество портфелей, каждый со своей величиной доходности. С точки зрения инвестора, наиболее привлекательными выглядят портфели с максимальной доходностью для фиксированного уровня риска или с наименьшим уровнем риска для фиксированной ожидаемой доходности. Такой набор портфелей составляет так называемую *границу эффективности*, которая будет рассматриваться далее.

## Оптимальный портфель

Описанная в этом разделе функция *минимизации* носит общий характер и позволяет учитывать параметрические ограничения, заданные равенствами, неравенствами и числовыми значениями.

Но сначала следует рассмотреть задачу *максимизации коэффициента Шарпа*. С формальной точки зрения для получения максимальной доходности и оптимального состава инвестиционного портфеля нужно минимизировать отрицательные значения коэффициента Шарпа. Ограничением в подобной задаче будет сумма параметров (весов), которая в нашем случае должна равняться 1. Ниже показано, как такая задача реализуется на Python с помощью функции `minimize()`<sup>4</sup>. Кроме того, все параметры (веса) должны находиться в диапазоне от 0 до 1. Необходимые значения передают функции `minimize()` в виде кортежа кортежей.

Единственное, чего не хватает для вызова функции оптимизации, — это список начальных параметров (начальное предположение о векторе весов). Пусть исходно все финансовые инструменты представлены в портфеле одинаковыми весами.

```
In [57]: import scipy.optimize as sco
```

```
In [58]: def min_func_sharpe(weights): ❶  
         return -port_ret(weights) / port_vol(weights) ❶
```

```
In [59]: cons = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) ❷
```

```
In [60]: bnds = tuple((0, 1) for x in range(noa)) ❸
```

```
In [61]: eweights = np.array(noa * [1. / noa,]) ❹  
         eweights ❹
```

```
Out[61]: array([0.25, 0.25, 0.25, 0.25])
```

```
In [62]: min_func_sharpe(eweights)  
Out[62]: -0.8436203363155397
```

- |                                     |                                   |
|-------------------------------------|-----------------------------------|
| ❶ Минимизируемая функция.           | ❸ Предельные значения параметров. |
| ❷ Ограничение, заданное равенством. | ❹ Вектор равных весов.            |

---

<sup>4</sup> Ограничение `np.sum(x) - 1` можно переписать как `np.sum(x) == 1`, поскольку в Python значение `True` равно 1, а значение `False` равно 0.



Функция минимизации возвращает не только список оптимальных значений параметров. Результаты ее работы сохраняются в объекте `opts`. Наибольший интерес для нас будут представлять сведения об оптимальном составе портфеля. Доступ к ним можно получить по ключу `x`.

```
In [63]: %%time
opts = sco.minimize(min_func_sharpe, eweights,
                    method='SLSQP', bounds=bnds,
                    constraints=cons) ❶
CPU times: user 67.6 ms, sys: 1.94 ms, total: 69.6 ms
Wall time: 75.2 ms
```

```
In [64]: opts ❷
Out[64]: fun: -0.8976673894052725
jac: array([ 8.96826386e-05, 8.30739737e-05,
            -2.45958567e-04, 1.92895532e-05])
message: 'Optimization terminated successfully.'
nfev: 36
nit: 6
njev: 6
status: 0
success: True
x: array([0.51191354, 0.19126414, 0.25454109,
          0.04228123])
```

```
In [65]: opts['x'].round(3) ❸
Out[65]: array([0.512, 0.191, 0.255, 0.042])
```

```
In [66]: port_ret(opts['x']).round(3) ❹
Out[66]: 0.161
```

```
In [67]: port_vol(opts['x']).round(3) ❺
Out[67]: 0.18
```

```
In [68]: port_ret(opts['x']) / port_vol(opts['x']) ❻
Out[68]: 0.8976673894052725
```

- ❶ Оптимизация (т.е. минимизация функции `min_func_sharpe()`).
- ❷ Результаты оптимизации.
- ❸ Оптимальные веса финансовых инструментов в портфеле.
- ❹ Результирующая доходность портфеля.

⑤ Результирующая волатильность портфеля.

⑥ Максимальный коэффициент Шарпа.

Теперь можно переходить к *минимизации дисперсии* (волатильности) портфеля.

```
In [69]: optv = sco.minimize(port_vol, eweights,  
                             method='SLSQP', bounds=bnds,  
                             constraints=cons) ❶
```

```
In [70]: optv
```

```
Out[70]: fun: 0.1094215526341138  
         jac: array([0.11098004, 0.10948556, 0.10939826,  
                    0.10944918])  
         message: 'Optimization terminated successfully.'  
         nfev: 54  
         nit: 9  
         njev: 9  
         status: 0  
         success: True  
         x: array([1.62630326e-18, 1.06170720e-03,  
                  5.43263079e-01, 4.55675214e-01])
```

```
In [71]: optv['x'].round(3)
```

```
Out[71]: array([0.    , 0.001, 0.543, 0.456])
```

```
In [72]: port_vol(optv['x']).round(3)
```

```
Out[72]: 0.109
```

```
In [73]: port_ret(optv['x']).round(3)
```

```
Out[73]: 0.06
```

```
In [74]: port_ret(optv['x']) / port_vol(optv['x'])
```

```
Out[74]: 0.5504173653075624
```

❶ Минимизация волатильности портфеля.

На этот раз портфель включает всего три финансовых инструмента, благодаря чему достигается минимальная волатильность (дисперсия).

## Граница эффективности

Нахождение всех возможных оптимальных портфелей — т.е. всех портфелей с минимальной волатильностью для заданного уровня доходности (или всех портфелей с максимальной доходностью для заданного уровня риска) — напоминает предыдущие задачи оптимизации. Единственное отличие заключается в том, что теперь нам нужно перебрать все варианты стартовых условий.

Наш подход будет состоять в последовательном фиксировании целевых уровней доходности и определении для каждого из них таких весов финансовых инструментов, которые обеспечивают минимизацию волатильности портфеля. В задаче оптимизации такой подход описывается двумя условиями. Одно из них касается целевого уровня доходности, `tret`, а второе, как и прежде, определяет сумму весов финансовых инструментов. Граничные значения для каждого параметра остаются прежними. В процессе итерирования по различным уровням доходности (`trets`) изменяется только одно из условий задачи минимизации. Таким образом, словарь ограничений обновляется на каждой итерации цикла.

```
In [75]: cons = ({'type': 'eq', 'fun': lambda x: port_ret(x) - tret},  
                {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) ❶
```

```
In [76]: bnds = tuple((0, 1) for x in weights)
```

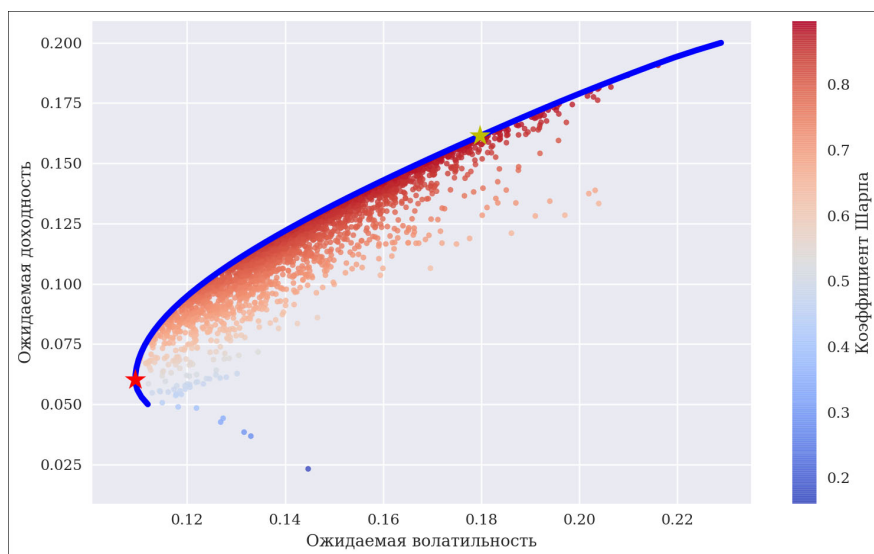
```
In [77]: %%time  
trets = np.linspace(0.05, 0.2, 50)  
tvols = []  
for tret in trets:  
    res = sco.minimize(port_vol, eweights, method='SLSQP',  
                      bounds=bnds, constraints=cons) ❷  
    tvols.append(res['fun'])  
tvols = np.array(tvols)  
CPU times: user 2.6 s, sys: 13.1 ms, total: 2.61 s  
Wall time: 2.66 s
```

- ❶ Два связанных ограничения границы эффективности.
- ❷ Минимизация волатильности портфеля для разных целевых уровней доходности.

Результаты такой оптимизации представлены на рис. 13.13. Сплошная кривая обозначает оптимальные портфели для заданных целевых уровней доходности. Отдельные точки — это, как и ранее, все возможные случайные порт-

фели. Дополнительно на кривую нанесены две звездочки: одна из них (слева внизу) определяет портфель с минимальной волатильностью (дисперсией), а вторая — портфель с максимальным коэффициентом Шарпа.

```
In [78]: plt.figure(figsize=(10, 6))
plt.scatter(pvols, pretts, c=pretts / pvols,
            marker='.', alpha=0.8, cmap='coolwarm')
plt.plot(tvols, tretts, 'b', lw=4.0)
plt.plot(port_vol(opts['x']), port_ret(opts['x']),
        'y*', markersize=15.0)
plt.plot(port_vol(optv['x']), port_ret(optv['x']),
        'r*', markersize=15.0)
plt.xlabel('Ожидаемая волатильность')
plt.ylabel('Ожидаемая доходность')
plt.colorbar(label='Коэффициент Шарпа')
```



*Рис. 13.13. Портфели с минимальным риском для заданных уровней доходности (граница эффективности)*

Граница эффективности состоит из всех оптимальных портфелей с более высокой доходностью, чем у портфеля с наименьшей дисперсией. Эти портфели оказываются более привлекательными для инвесторов с точки зрения ожидаемой доходности для заданного уровня риска.

## Линия рынка капиталов

Наряду с рисковыми финансовыми инструментами, такими как акции или биржевые товары (например, золото), всегда существует универсальная возможность безрискового инвестирования — в валюту или денежные средства. В идеальном мире деньги, которые хранятся на счете в крупном банке, могут считаться безрисковым активом (их сохранность гарантируется государственными программами страхования вкладов). Основной недостаток подобных безрисковых инвестиций — это чрезвычайно низкая доходность, иногда близкая к нулю.

В то же время добавление такого безрискового актива в портфель сильно повышает его инвестиционную привлекательность. Основная идея заключается в том, что инвесторы сначала определяют эффективный портфель рискованных активов, а затем добавляют к нему безрисковый актив. Корректируя долю капитала, инвестируемого в безрисковый актив, можно достичь любого соотношения риск/доходность вдоль прямой, проложенной от точки безрискового актива к точке эффективного портфеля.

Но какой из всех возможных эффективных портфелей будет считаться оптимальным с точки зрения инвестирования? Тот, для которого касательная к границе эффективности проходит непосредственно через точку безрискового портфеля. Рассмотрим, к примеру, безрисковую процентную ставку  $r_f = 0,01$ . Оптимальным для нее будет портфель, для которого касательная к границе эффективности проходит через точку  $(\sigma_f, r_f) = (0; 0,01)$  на графике соотношения риска и доходности.

Для дальнейших расчетов мы создадим две функции, которые аппроксимируют функцию границы эффективности и ее первую производную. Такое дифференцируемое функциональное приближение достигается за счет применения кубических сплайнов (см. главу 11). В сплайновой интерполяции участвуют только портфели, относящиеся к границе эффективности. Это позволяет определить непрерывно дифференцируемую функцию границы эффективности  $f(x)$  и ее первую производную  $df(x)$ .

```
In [79]: import scipy.interpolate as sci
```

```
In [80]: ind = np.argmin(tvols) ❶  
         evols = tvols[ind:] ❷  
         erets = trets[ind:] ❷
```

```
In [81]: tck = sci.splrep(evols, erets) ❸
```

```
In [82]: def f(x):
        ''' Функция границы эффективности
            (аппроксимация сплайнами) '''
        return sci.splev(x, tck, deg=0)
        def df(x):
            ''' Первая производная функции
                границы эффективности '''
            return sci.splev(x, tck, deg=1)
```

- ❶ Индекс портфеля с минимальной волатильностью
- ❷ Соответствующие значения волатильности и доходности.
- ❸ Интерполяция кубическими сплайнами по указанным значениям.

Теперь необходимо получить линейную функцию  $t(x) = a + bx$ , представляющую собой касательную к границе эффективности, которая проходит через точку безрискового актива. Такое поведение линейной функции  $t(x)$  описывается следующими тремя условиями (уравнение 13.4).

**Уравнение 13.4. Математические условия для построения линии рынка капиталов**

$$\begin{aligned} t(x) &= a + bx, \\ t(0) &= r_f \Leftrightarrow a = r_f, \\ t(x) &= f(x) \Leftrightarrow a + bx = f(x), \\ t'(x) &= f'(x) \Leftrightarrow b = f'(x). \end{aligned}$$

Поскольку не существует замкнутой формулы для описания функции границы эффективности или ее первой производной, систему уравнений 13.4 придется решать численными методами. Мы определим функцию Python, которая возвращает решения всех трех уравнений для заданного набора параметров  $p = (a, b, x)$ .

Такую систему уравнений способна решить функция `sco.fsolve()` из пакета `scipy.optimize`. Ей передается функция `equations()`, описывающая систему уравнений, а также начальный набор параметров. Учтите, что результат оптимизации может зависеть от начальных параметров, поэтому их следует выбирать осмотрительно — обычно требуемая комбинация подбирается методом проб и ошибок.

```
In [83]: def equations(p, rf=0.01):
        eq1 = rf - p[0] ❶
        eq2 = rf + p[1] * p[2] - f(p[2]) ❶
```

```
eq3 = p[1] - df(p[2]) ❶
return eq1, eq2, eq3
```

```
In [84]: opt = sco.fsolve(equations, [0.01, 0.5, 0.15]) ❷
```

```
In [85]: opt ❸
```

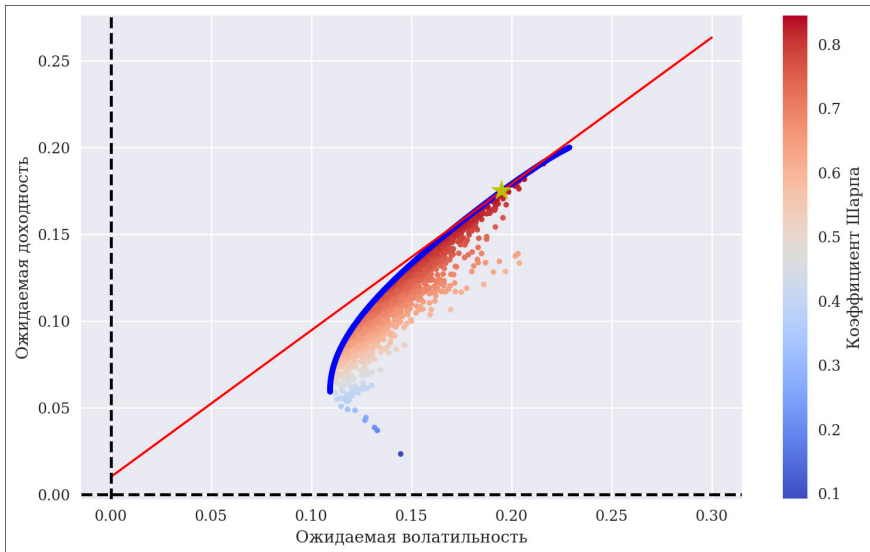
```
Out[85]: array([0.01      , 0.84470952, 0.19525391])
```

```
In [86]: np.round(equations(opt), 6) ❹
```

```
Out[86]: array([ 0.,  0., -0.])
```

- ❶ Уравнения, описывающие линию рынка капиталов.
- ❷ Решение системы уравнений для заданных начальных параметров.
- ❸ Оптимальные значения параметров.
- ❹ Все решения уравнения равны нулю.

На рис. 13.14 показан результат в графическом виде. Звездочкой помечен оптимальный портфель на границе эффективности, для которого касательная проходит через точку безрискового актива (0,  $r_f = 0,01$ ).



**Рис. 13.14.** Линия рынка капиталов и оптимальный портфель (обозначен звездочкой) для безрисковой ставки 1%

```
In [87]: plt.figure(figsize=(10, 6))
plt.scatter(pvols, pretss, c=(pretss - 0.01) / pvols,
            marker='.', cmap='coolwarm')
plt.plot(evols, erets, 'b', lw=4.0)
cx = np.linspace(0.0, 0.3)
plt.plot(cx, opt[0] + opt[1] * cx, 'r', lw=1.5)
plt.plot(opt[2], f(opt[2]), 'y*', markersize=15.0)
plt.grid(True)
plt.axhline(0, color='k', ls='--', lw=2.0)
plt.axvline(0, color='k', ls='--', lw=2.0)
plt.xlabel('Ожидаемая волатильность')
plt.ylabel('Ожидаемая доходность')
plt.colorbar(label='Коэффициент Шарпа')
```

Веса финансовых инструментов в оптимальном портфеле рассчитываются так, как показано ниже. Заметьте, что в портфель включаются только три из них.

```
In [88]: cons = ({'type': 'eq', 'fun': lambda x:
                port_ret(x) - f(opt[2])},
                {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) ❶
res = sco.minimize(port_vol, eweights, method='SLSQP',
                  bounds=bnds, constraints=cons)
```

```
In [89]: res['x'].round(3) ❷
Out[89]: array([0.59 , 0.221, 0.189, 0.   ])
```

```
In [90]: port_ret(res['x'])
Out[90]: 0.1749328414905194
```

```
In [91]: port_vol(res['x'])
Out[91]: 0.19525371793918325
```

```
In [92]: port_ret(res['x']) / port_vol(res['x'])
Out[92]: 0.8959257899765407
```

- ❶ Связанные ограничения для оптимального портфеля (обозначен звездочкой на рис. 13.14).
- ❷ Веса финансовых инструментов для данного портфеля.



# Байесовская статистика

В наши дни байесовская статистика находит широкое применение в эмпирических финансовых расчетах. Одной главы, конечно же, будет недостаточно для того, чтобы подробно раскрыть данную тему. Более детальную информацию можно найти в книгах Гевеке [5] и Рачева [10].

## Формула Байеса

В финансовых вычислениях наибольшее распространение получила *диахроническая интерпретация* теоремы Байеса. Согласно ей рано или поздно появляется новая информация о представляющих интерес переменных или параметрах, таких как средняя доходность временного ряда. В математическом виде формула Байеса представлена уравнением 13.5.

### Уравнение 13.5. Формула Байеса

$$p(H|D) = \frac{p(H)p(D|H)}{p(D)},$$

где  $H$  — это событие (гипотеза), а  $D$  — данные, полученные в результате эксперимента или практических измерений<sup>5</sup>. Смысл остальных обозначений таков:

- $p(H)$  — *априорная* вероятность гипотезы  $H$ ;
- $p(D)$  — вероятность получения данных, подтверждающих любую гипотезу (*нормализующая константа*);
- $p(D|H)$  — *правдоподобие* (вероятность получения данных, подтверждающих гипотезу  $H$ );
- $p(H|D)$  — *апостериорная* вероятность гипотезы  $H$  после получения данных.

Рассмотрим простой пример. Имеются два контейнера,  $B_1$  и  $B_2$ . В контейнере  $B_1$  находится 30 черных и 60 красных шаров, а в контейнере  $B_2$  — 60 черных и 30 красных шаров. Из одного контейнера случайным образом вынимают шар. Предположим, что он *черного* цвета. Рассчитаем вероятности следующих гипотез:

- $H_1$  — шар извлекается из контейнера  $B_1$ .
- $H_2$  — шар извлекается из контейнера  $B_2$ .

---

<sup>5</sup> Эти и другие положения байесовской статистики описаны в книге Дауни [4].

Перед извлечением шара обе гипотезы равновероятные. Как только из одного контейнера извлекается черный шар, вероятности гипотез изменяются по формуле Байеса. Рассчитаем обновленную вероятность гипотезы  $H_1$ .

- Априорная вероятность  $p(H_1) = \frac{1}{2}$ .
- Нормализующая константа  $p(D) = \frac{1}{2} \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{2}{3} = \frac{1}{2}$ .
- Правдоподобие  $p(D|H_1) = \frac{1}{3}$ .

Следовательно, обновленная вероятность гипотезы  $H_1$  рассчитывается так:

$$p(H_1|D) = \frac{\frac{1}{2} \cdot \frac{1}{3}}{\frac{1}{2}} = \frac{1}{3}.$$

В этом результате нет ничего удивительного. Исходно вероятность извлечения черного шара из контейнера  $B_2$  вдвое выше вероятности его извлечения из контейнера  $B_1$ . После извлечения черного шара обновленная вероятность гипотезы  $H_2$  будет равна  $p(H_2|D) = \frac{2}{3}$ , что по-прежнему вдвое больше вероятности гипотезы  $H_1$ .

## Байесовская регрессия

Пакет PyMC3, являющийся частью экосистемы Python, содержит средства байесовской статистики и вероятностного программирования.

Рассмотрим пример зашумленных данных, сгруппированных вокруг прямой линии<sup>6</sup>. Сначала реализуем простую линейную регрессию по методу наименьших квадратов (см. главу 11). Результат представлен на рис. 13.15.

```
In [1]: import numpy as np
import pandas as pd
import datetime as dt
from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
np.random.seed(1000)
%matplotlib inline
```

---

<sup>6</sup> Этот пример был предложен Томасом Виецки, одним из разработчиков пакета PyMC3.

```

In [3]: x = np.linspace(0, 10, 500)
        y = 4 + 2 * x + np.random.standard_normal(len(x)) * 2

In [4]: reg = np.polyfit(x, y, 1)

In [5]: reg
Out[5]: array([2.03384161, 3.77649234])

In [6]: plt.figure(figsize=(10, 6))
        plt.scatter(x, y, c=y, marker='v', cmap='coolwarm')
        plt.plot(x, reg[1] + reg[0] * x, lw=2.0)
        plt.colorbar()
        plt.xlabel('x')
        plt.ylabel('y')

```



*Рис. 13.15. Точки выборки и регрессионная прямая*

Применяя метод наименьших квадратов, можно получить два основных параметра (наклон и сдвиг) регрессионной прямой. Обратите внимание на то, что в данном случае коэффициент при факторе наибольшего порядка (наклон регрессионной прямой) соответствует индексу 0, а константа (сдвиг) — индексу 1. Исходные значения параметров 2 и 4 не удалось в точности воспроизвести из-за зашумленности набора данных.

Теперь выполним байесовскую регрессию с помощью пакета РумСЗ. Предполагается, что параметры имеют определенное распределение. Например, рассмотрим уравнение регрессионной прямой  $\hat{y}(x) = \alpha + \beta x$ . Мы предполагаем следующие *априорные* характеристики:

- коэффициент  $\alpha$  имеет нормальное распределение со средним 0 и стандартным отклонением 20;
- коэффициент  $\beta$  имеет нормальное распределение со средним 0 и стандартным отклонением 10.

Для функции *правдоподобия* ожидается нормальное распределение со средним  $\hat{y}(x)$  и стандартным отклонением, равномерно распределенным в диапазоне от 0 до 10.

Ключевой особенностью байесовской регрессии является применение *методов Монте-Карло с марковскими цепями* (Markov chain Monte Carlo — MCMC)<sup>7</sup>. Это напоминает предыдущий пример с многократным извлечением шаров из контейнеров, только здесь все делается в более систематизированном и автоматизированном виде.

Для выборки нам потребуются три функции.

- `find_MAP()`. Находит начальную точку для алгоритма выборки, определяя локальный максимум апостериорной точки.
- `NUTS()`. Реализует алгоритм NUTS (No-U-Turn Sampler) с учетом предполагаемых априорных значений.
- `sample()`. Выполняет заданное число выборок с начальным значением, полученным от функции `find_MAP()`, и оптимальным размером шага, который был определен алгоритмом NUTS.

Все три функции инкапсулированы в объекте `Model` пакета РумСЗ и вызываются в блоке `with`.

```
In [8]: import rumc3 as pm
```

```
In [9]: %%time
```

```
    with pm.Model() as model:
```

```
        # Модель
```

```
        alpha = pm.Normal('alpha', mu=0, sd=20) ❶
```

```
        beta = pm.Normal('beta', mu=0, sd=10) ❶
```

---

<sup>7</sup> Все применяемые в книге алгоритмы, основанные на методе Монте-Карло, генерируют так называемые *марковские цепи*, поскольку значение на следующем шаге зависит только от текущего состояния процесса и не зависит ни от каких предыдущих состояний.

```

sigma = pm.Uniform('sigma', lower=0, upper=10) ❶
y_est = alpha + beta * x ❷
likelihood = pm.Normal('y', mu=y_est, sd=sigma,
                       observed=y) ❸

# Байесовский вывод
start = pm.find_MAP() ❹
step = pm.NUTS() ❺
trace = pm.sample(100, tune=1000, start=start,
                  progressbar=True, verbose=False) ❻
logp = -1,067.8, ||grad|| = 60.354: 100%|██████████| 28/28
[00:00<00:00, 474.70it/s]
Only 100 samples in chain.
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sigma, beta, alpha]
Sampling 2 chains: 100%|██████████| 2200/2200
[00:03<00:00, 690.96draws/s]
CPU times: user 6.2 s, sys: 1.72 s, total: 7.92 s
Wall time: 1min 28s

```

In [10]: pm.summary(trace) ❼

```

Out[10]:

```

	mean	sd	mc_error	hpd_2.5	hpd_97.5	\
<b>alpha</b>	3.764027	0.174796	0.013177	3.431739	4.070091	
<b>beta</b>	2.036318	0.030519	0.002230	1.986874	2.094008	
<b>sigma</b>	2.010398	0.058663	0.004517	1.904395	2.138187	

	n_eff	Rhat
<b>alpha</b>	152.446951	0.996281
<b>beta</b>	106.505590	0.999155
<b>sigma</b>	188.643293	0.998547

In [11]: trace[0] ❽

```

Out[11]: {'alpha': 3.9303300798212444,
          'beta': 2.0020264758995463,
          'sigma_interval__': -1.3519315719461853,
          'sigma': 2.0555476283253156}

```

❶ Задание априорных параметров.

❷ Линейная регрессия.

- ③ Функция правдоподобия.
- ④ Нахождение начального значения с помощью функции оптимизации.
- ⑤ Инициализация алгоритма MCMC.
- ⑥ Получение апостериорных выборок данных по алгоритму NUTS.
- ⑦ Отображение статистической сводки.
- ⑧ Оценочные значения для первой выборки.

Отображаемые оценки достаточно близки к исходным значениям (4, 2, 2). Но программа получает намного больше оценок, чем было выведено на экран. Для их анализа лучше всего воспользоваться *трассировочным графиком* (рис. 13.16), на котором можно увидеть апостериорное распределение различных параметров, а также все одиночные оценки по каждой выборке. Апостериорное распределение позволяет получить представление о неопределенности оценок.

```
In [12]: pm.traceplot(trace, lines=['alpha': 4, 'beta': 2, 'sigma': 2]);
```

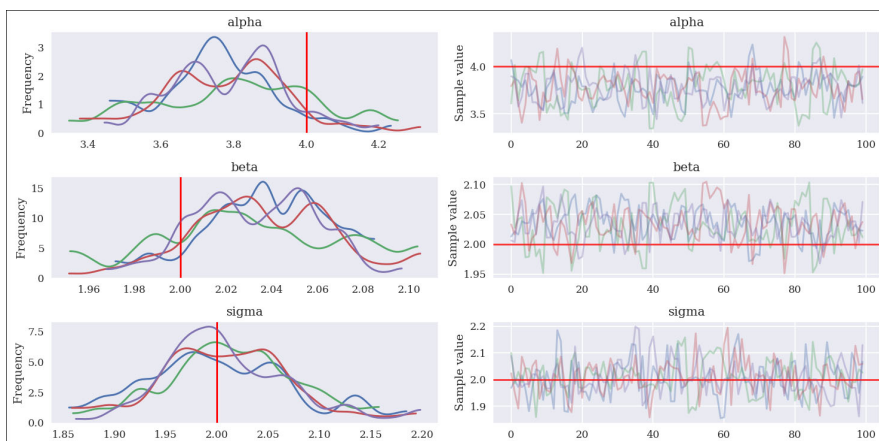


Рис. 13.16. Апостериорные распределения и трассировочные графики

Если брать в расчет только параметры `alpha` и `beta`, то можно получить целый набор регрессионных прямых (рис. 13.17).

```
In [13]: plt.figure(figsize=(10, 6))
plt.scatter(x, y, c=y, marker='v', cmap='coolwarm')
plt.colorbar()
plt.xlabel('x')
```

```
plt.ylabel('y')
for i in range(len(trace)):
    plt.plot(x, trace['alpha'][i] + trace['beta'][i] * x) ❶
```

❶ Построение отдельной регрессионной прямой.

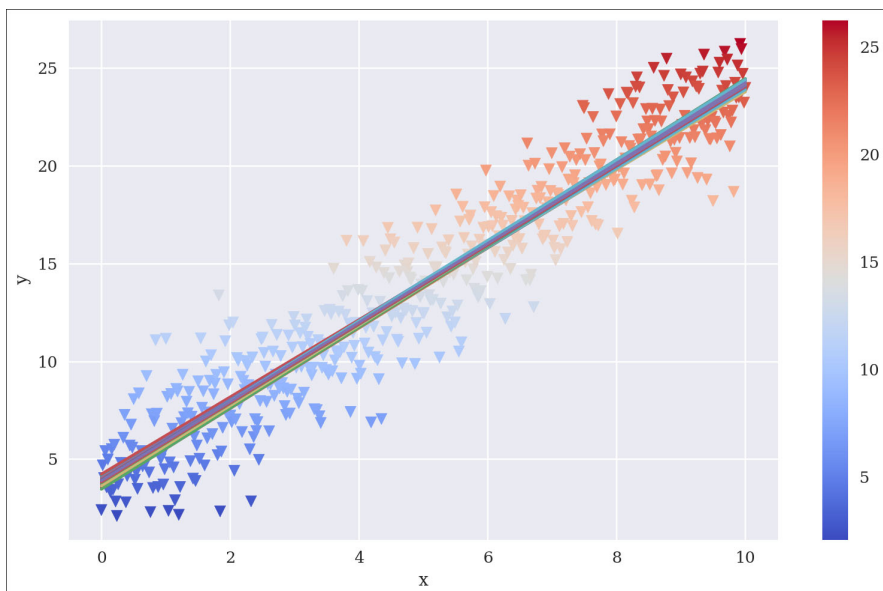


Рис. 13.17. Регрессионные прямые для различных оценочных значений

## Два финансовых инструмента

Успешно выполнив регрессионные расчеты на фиктивных данных с помощью пакета PyMC3, мы можем переходить к анализу реальных финансовых данных. В следующем примере используются данные финансовых временных рядов двух биржевых инвестиционных фондов: GLD и GDX (рис. 13.18).

```
In [14]: raw = pd.read_csv('../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True)
```

```
In [15]: data = raw[['GDX', 'GLD']].dropna()
```

```
In [16]: data = data / data.iloc[0] ❶
```

```
In [17]: data.info()
<class 'pandas.core.frame.DataFrame'>
```

```
DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
Data columns (total 2 columns):
GDX      2138 non-null float64
GLD      2138 non-null float64
dtypes: float64(2)
memory usage: 50.1 KB
```

```
In [18]: data.ix[-1] / data.ix[0] - 1 ❷
```

```
Out[18]: GDX      -0.532383
          GLD       0.080601
          dtype: float64
```

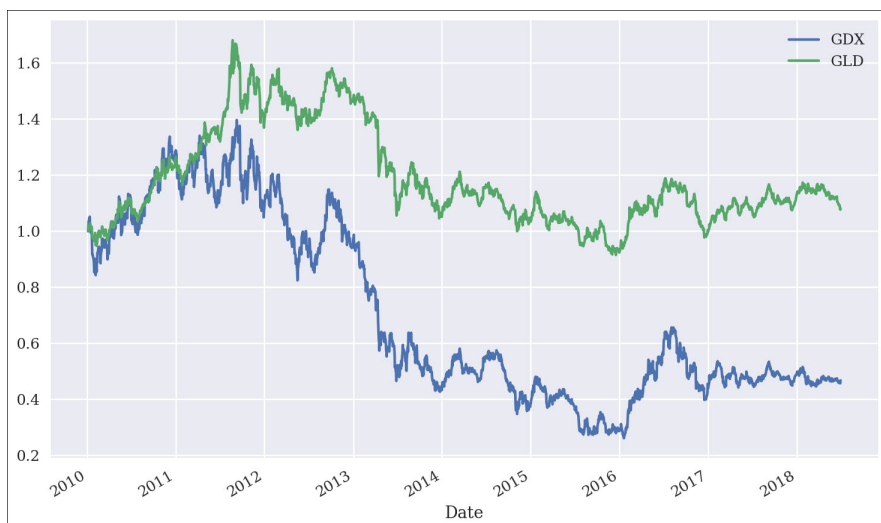
```
In [19]: data.corr() ❸
```

```
Out[19]:
```

	GDX	GLD
GDX	1.000000	0.71539
GLD	0.71539	1.000000

```
In [20]: data.plot(figsize=(10, 6));
```

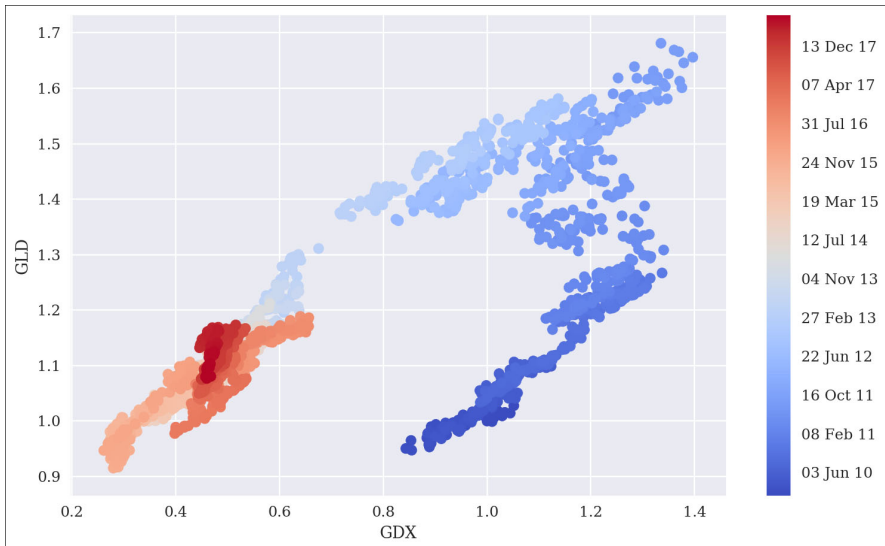
- ❶ Нормализация данных к начальному значению 1.
- ❷ Вычисление относительной доходности.
- ❸ Вычисление корреляции между двумя финансовыми инструментами.



*Рис. 13.18. Изменение нормализованных цен на акции GLD и GDX во времени*



В следующем коде строится диаграмма рассеяния, которая визуализирует даты, относящиеся к отдельным точкам данных. Для этого объект `DatetimeIndex` класса `DataFrame` приводится к формату даты, принятому в библиотеке `matplotlib`. На рис. 13.19 показана точечная диаграмма, осями которой служат временные ряды GLD и GDХ, а на вертикальной шкале разными оттенками обозначаются даты каждой пары значений<sup>8</sup>.



*Рис. 13.19. Диаграмма рассеяния цен на акции GLD относительно котировок акций GDХ*

```
In [21]: data.index[:3]
Out[21]: DatetimeIndex(['2010-01-04', '2010-01-05', '2010-01-06'],
                        dtype='datetime64[ns]', name='Date', freq=None)

In [22]: mpl_dates = mpl.dates.date2num(data.index.to_pydatetime()) ❶
          mpl_dates[:3]
Out[22]: array([733776., 733777., 733778.])

In [23]: plt.figure(figsize=(10, 6))
          plt.scatter(data['GDX'], data['GLD'], c=mpl_dates,
                      marker='o', cmap='coolwarm')
          plt.xlabel('GDX')
```

<sup>8</sup> Заметим, что все приводимые здесь диаграммы основаны на нормализованных данных о ценах, а не на данных о доходности, как было бы привычнее для финансовых приложений.

```
plt.ylabel('GLD')
plt.colorbar(ticks=mpl.dates.DayLocator(interval=250),
             format=mpl.dates.DateFormatter('%d %b %y')); ❷
```

- ❶ Преобразование объекта `DatetimeIndex` в даты формата `matplotlib`.
- ❷ Настройка цветовой шкалы для дат.

В следующем коде реализуется алгоритм байесовской регрессии для двух наших временных рядов. Параметризация здесь почти та же, что и в примере с фиктивными данными. На рис. 13.20 показаны результаты, полученные при выборке данных по методу МСМС с учетом предположений об априорном распределении вероятностей трех оцениваемых параметров.

```
In [24]: with pm.Model() as model:
        alpha = pm.Normal('alpha', mu=0, sd=20)
        beta = pm.Normal('beta', mu=0, sd=20)
        sigma = pm.Uniform('sigma', lower=0, upper=50)

        y_est = alpha + beta * data['GDX'].values

        likelihood = pm.Normal('GLD', mu=y_est, sd=sigma,
                               observed=data['GLD'].values)

        start = pm.find_MAP()
        step = pm.NUTS()
        trace = pm.sample(250, tune=2000, start=start,
                          progressbar=True)

logp = 1,493.7, ||grad|| = 188.29: 100%|██████████| 27/27
[00:00<00:00, 1609.34it/s]
Only 250 samples in chain.
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sigma, beta, alpha]
Sampling 2 chains: 100%|██████████| 4500/4500
[00:09<00:00, 465.07draws/s]
The estimated number of effective samples is smaller than
200 for some parameters.
```

```
In [25]: pm.summary(trace)
```

Out[25]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5	\
<b>alpha</b>	0.913335	0.005983	0.000356	0.901586	0.924714	
<b>beta</b>	0.385394	0.007746	0.000461	0.369154	0.398291	
<b>sigma</b>	0.119484	0.001964	0.000098	0.115305	0.123315	

	n_eff	Rhat
<b>alpha</b>	184.264900	1.001855
<b>beta</b>	215.477738	1.001570
<b>sigma</b>	312.260213	1.005246

In [26]: fig = pm.traceplot(trace)

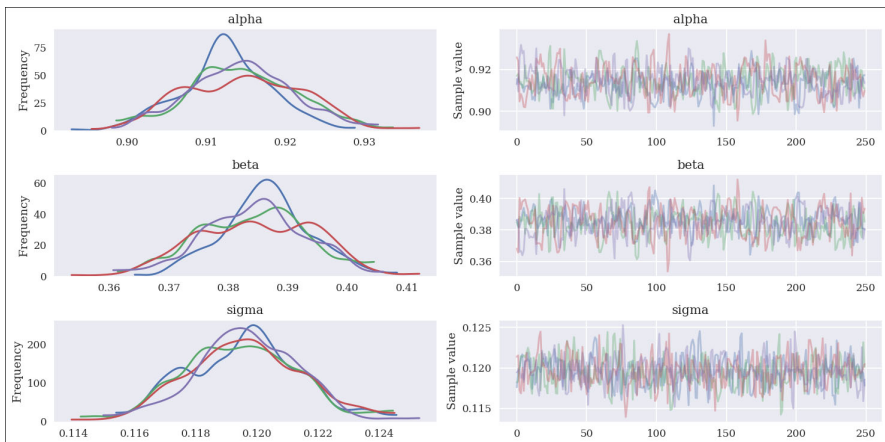
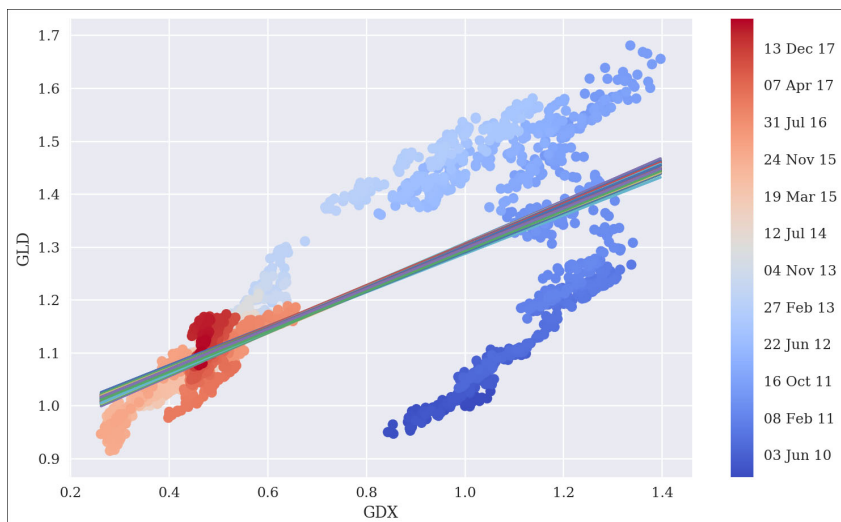


Рис. 13.20. Апостериорные распределения и трассировочные графики для индексов GDX и GLD

На рис. 13.21 показаны регрессионные прямые, добавленные на полученную ранее диаграмму рассеяния. Как видите, все они расположены очень близко.

```
In [27]: plt.figure(figsize=(10, 6))
plt.scatter(data['GDX'], data['GLD'], c=mpl_dates,
            marker='o', cmap='coolwarm')
plt.xlabel('GDX')
plt.ylabel('GLD')
for i in range(len(trace)):
    plt.plot(data['GDX'], trace['alpha'][i] +
             trace['beta'][i] * data['GDX'])
plt.colorbar(ticks=mpl.dates.DayLocator(interval=250),
            format=mpl.dates.DateFormatter('%d %b %y'));
```



*Рис. 13.21. Многочисленные байесовские регрессионные прямые для индексов GDX и GLD*

Это наглядно показывает основной недостаток регрессионного подхода: он не учитывает эволюцию данных во времени. В нем последние данные обрабатываются точно так же, как и самые первые.

## Обновление оценочных значений со временем

Как уже говорилось ранее, в финансовых вычислениях байесовский подход оказывается наиболее полезным в диахронической интерпретации, когда новые данные, появляющиеся со временем, позволяют улучшать результаты регрессии и получать более точные оценки в процессе обновления или обучения модели.

В качестве примера предположим, что наши регрессионные параметры не только имеют случайную природу и характеризуются каким-то распределением, но и подвержены случайному блужданию. Это то же самое обобщение, которое применяется в финансовой теории при переходе от случайных переменных к стохастическим процессам (т.е. упорядоченным последовательностям случайных величин).

Нам придется определить новую модель РумСЗ, в которой значения параметров описываются как случайные блуждания. Сначала нужно задать распределения параметров, а затем описать случайные блуждания для параметров  $\alpha$  и  $\beta$ . Для повышения производительности модели зададим общие коэффициенты для 50 временных точек данных.

```
In [28]: from pymc3.distributions.timeseries import GaussianRandomWalk
```

```
In [29]: subsample_alpha = 50  
        subsample_beta = 50
```

```
In [30]: model_randomwalk = pm.Model()  
        with model_randomwalk:  
            sigma_alpha = pm.Exponential('sig_alpha', 1. /  
                                         02, testval=.1) ❶  
            sigma_beta = pm.Exponential('sig_beta', 1. / .02,  
                                         testval=.1) ❶  
            alpha = GaussianRandomWalk('alpha', sigma_alpha ** -2,  
                                       shape=int(len(data) /  
                                       subsample_alpha)) ❷  
            beta = GaussianRandomWalk('beta', sigma_beta ** -2,  
                                       shape=int(len(data) /  
                                       subsample_beta)) ❷  
            alpha_r = np.repeat(alpha, subsample_alpha) ❸  
            beta_r = np.repeat(beta, subsample_beta) ❸  
            regression = alpha_r + beta_r * \  
                        data['GDX'].values[:2100] ❹  
            sd = pm.Uniform('sd', 0, 20) ❺  
            likelihood = pm.Normal('GLD', mu=regression, sd=sd,  
                                   observed=data['GLD'].values[:2100]) ❻
```

- ❶ Задание априорных значений для параметров случайного блуждания.
- ❷ Модели случайных блужданий.
- ❸ Приведение векторов параметров к длительности интервала.
- ❹ Определение регрессионной модели.
- ❺ Априорные значения для стандартного отклонения.
- ❻ Определение функции правдоподобия на основе результатов регрессии.

Использование случайных блужданий вместо случайных переменных приводит к усложнению параметризации модели. В то же время процедура байесовского вывода по методу МСМС остается практически той же. Но не забывайте, что объем вычислений существенно увеличивается, поскольку теперь алгоритму нужно оценивать в каждой выборке все параметры случайного блуждания —  $1\,950 / 50 = 39$  комбинаций параметров вместо 1, как раньше.

```
In [31]: %time  
        import scipy.optimize as sco
```

```

with model_randomwalk:
    start = pm.find_MAP(vars=[alpha, beta],
                        fmin=sco.fmin_l_bfgs_b)
    step = pm.NUTS(scaling=start)
    trace_rw = pm.sample(250, tune=1000, start=start,
                        progressbar=True)
logp = -6,657: 2%| | 82/5000 [00:00<00:08, 550.29it/s]
Only 250 samples in chain.
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sd, beta, alpha, sig_beta, sig_alpha]
Sampling 2 chains: 100%|██████████| 2500/2500
[02:48<00:00, 8.59draws/s]
CPU times: user 27.5 s, sys: 3.68 s, total: 31.2 s
Wall time: 5min 3s

```

In [32]: `pm.summary(trace_rw).head()` ❶

Out[32]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5	\
<b>alpha__0</b>	0.673846	0.040224	0.001376	0.592655	0.753034	
<b>alpha__1</b>	0.424819	0.041257	0.001618	0.348102	0.509757	
<b>alpha__2</b>	0.456817	0.057200	0.002011	0.321125	0.553173	
<b>alpha__3</b>	0.268148	0.044879	0.001725	0.182744	0.352197	
<b>alpha__4</b>	0.651465	0.057472	0.002197	0.544076	0.761216	

	n_eff	Rhat
<b>alpha__0</b>	1004.616544	0.998637
<b>alpha__1</b>	804.760648	0.999540
<b>alpha__2</b>	800.225916	0.998075
<b>alpha__3</b>	724.967532	0.998995
<b>alpha__4</b>	978.073246	0.998060

❶ Статистическая сводка по интервалу (первые пять параметров `alpha`).

На рис. 13.22 можно увидеть, как меняются регрессионные параметры `alpha` и `beta` во времени.

In [33]: `sh = np.shape(trace_rw['alpha'])` ❶

`sh` ❶

Out[33]: `(500, 42)`

In [34]: `part_dates = np.linspace(min(mpl_dates),  
max(mpl_dates), sh[1])` ❷

```
In [35]: index = [dt.datetime.fromordinal(int(date)) for  
              date in part_dates] ❷
```

```
In [36]: alpha = {'alpha_%i' % i: v for i, v in  
                enumerate(trace_rw['alpha']) if i < 20} ❸
```

```
In [37]: beta = {'beta_%i' % i: v for i, v in  
               enumerate(trace_rw['beta']) if i < 20} ❸
```

```
In [38]: df_alpha = pd.DataFrame(alpha, index=index) ❸
```

```
In [39]: df_beta = pd.DataFrame(beta, index=index) ❸
```

```
In [40]: ax = df_alpha.plot(color='b', style='-.', legend=False,  
                           lw=0.7, figsize=(10, 6))  
        df_beta.plot(color='r', style='-.', legend=False,  
                     lw=0.7, ax=ax)  
        plt.ylabel('alpha/beta');
```

- ❶ Форма объекта, содержащего оценочные значения параметра.
- ❷ Создание списка дат, который должен соответствовать количеству интервалов.
- ❸ Занесение соответствующих временных рядов в объекты DataFrame.

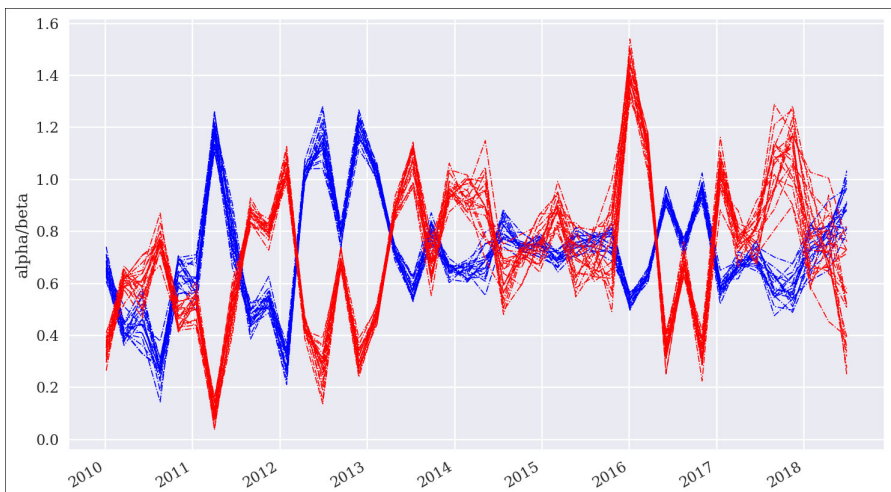


Рис. 13.22. Изменение оценочных значений параметров во времени



## Абсолютные цены и относительные доходности

В этом разделе мы работаем с нормализованными данными о ценах. Это сделано исключительно ради наглядности, поскольку соответствующие графики легче понимать и анализировать. В реальных финансовых приложениях чаще приходится иметь дело с данными о доходности, гарантирующими стационарность временных рядов.

Используя средние значения параметров  $\alpha$  и  $\beta$ , можно отслеживать изменение регрессии во времени. На рис. 13.23 отображаются 39 регрессионных прямых, полученных для разных пар параметров. Очевидно, что каждое последующее обновление данных существенно повышает точность регрессионной оценки (для текущих, т.е. самых последних, данных). Другими словами, для каждого периода времени требуется собственный регрессионный анализ.

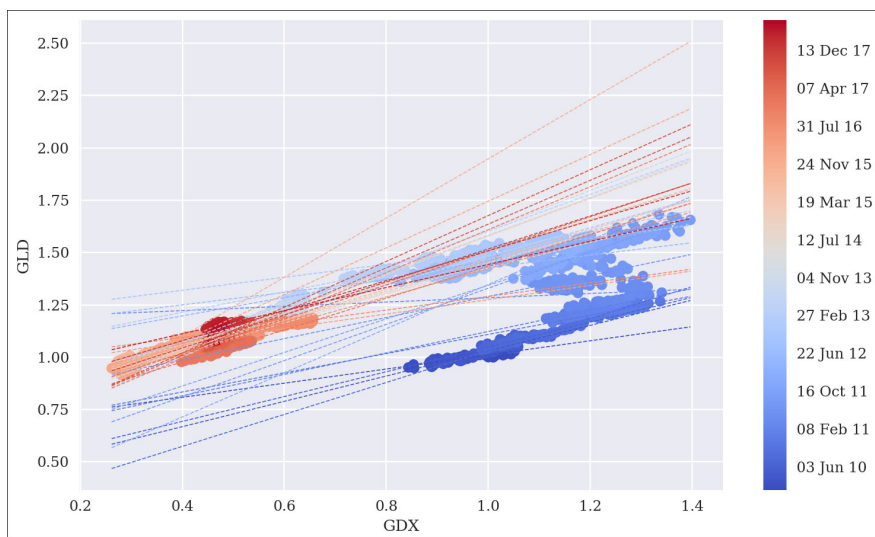


Рис. 13.23. Диаграмма рассеяния с регрессионными прямыми, получаемыми для разных временных интервалов (обновленные оценки)

```
In [41]: plt.figure(figsize=(10, 6))
plt.scatter(data['GDX'], data['GLD'], c=mpl_dates,
            marker='o', cmap='coolwarm')
plt.colorbar(ticks=mpl.dates.DayLocator(interval=250),
            format=mpl.dates.DateFormatter('%d %b %y'))
plt.xlabel('GDX')
plt.ylabel('GLD')
x = np.linspace(min(data['GDX']), max(data['GDX']))
```



```

for i in range(sh[1]): ❶
    alpha_rw = np.mean(trace_rw['alpha'].T[i])
    beta_rw = np.mean(trace_rw['beta'].T[i])
    plt.plot(x, alpha_rw + beta_rw * x, '--', lw=0.7,
             color=plt.cm.coolwarm(i / sh[1]))

```

- ❶ Построение регрессионных прямых для всех временных интервалов длительностью 50.

На этом мы завершаем рассмотрение байесовской статистики. В Python байесовские методы и алгоритмы вероятностного программирования реализованы в пакете PyMC3. Особую популярность в финансовой математике приобрела байесовская регрессия, которая остается важным инструментом финансовых вычислений.

## Машинное обучение

*Машинное обучение* — популярная тема не только в финансах, но и во многих других областях. Лучше всего его важность описывается следующей цитатой.

*Эконометрика хороша как академическая дисциплина, но для достижения практических результатов требуется машинное обучение.*

[8]

Машинное обучение — это целый класс алгоритмов, способных *самостоятельно* выявлять определенные закономерности и взаимосвязи в исходном наборе необработанных данных. Математические и статистические основы алгоритмов машинного обучения, а также особенности их реализации и практического применения подробно описаны в специализированных изданиях, перечисленных в конце главы. В частности, в книге Алпайдина [2] содержится введение в машинное обучение и дан нетехнический обзор основных алгоритмов.

В этом разделе нас будет интересовать исключительно практическая сторона вопроса. Мы поговорим о реализации определенных алгоритмов и познакомимся с рядом методик, которые будут рассматриваться в главе 15. Впрочем, они находят применение не только в алгоритмической торговле, но и во многих других финансовых областях. Нас ждет знакомство с двумя типами алгоритмов: обучение *без учителя* и обучение *с учителем*.

Библиотека Scikit-learn (<http://scikit-learn.org/>) — один из самых популярных пакетов машинного обучения на Python. Она не только предлагает

реализацию множества алгоритмов, но и содержит немало полезных инструментов обработки данных для задач машинного обучения. В примерах раздела мы в основном будем работать с ней. Также мы рассмотрим применение библиотеки TensorFlow (<http://tensorflow.org/>) в контексте работы с глубокими нейронными сетями.

Познакомиться с алгоритмами машинного обучения на Python можно в книге Вандер Пласа [11]. У Элбона [1] предлагаются готовые рецепты решения типичных задач машинного обучения, в основном на Python и Scikit-learn.

## Обучение без учителя

*Обучение без учителя* представляет собой концепцию машинного обучения, согласно которой алгоритм способен выявлять закономерности в необработанных данных без какого-либо вмешательства со стороны экспериментатора. Один из таких алгоритмов — *кластеризация методом  $k$ -средних*, когда исходный набор данных разбивается на отдельные поднаборы (кластеры), снабжаемые *метками* (“кластер 1”, “кластер 2” и т.д.). Другой популярный алгоритм — *смесь гауссиан* (гауссова смесь распределений)<sup>9</sup>.

### Данные

Помимо прочего, библиотека Scikit-learn позволяет создавать тестовые выборки для различных алгоритмов машинного обучения. Ниже мы создадим тестовый набор для алгоритма кластеризации по методу  $k$ -средних.

В первую очередь необходимо импортировать требуемые пакеты и задать конфигурационные настройки.

```
In [1]: import numpy as np
import pandas as pd
import datetime as dt
from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
np.random.seed(1000)
np.set_printoptions(suppress=True, precision=4)
%matplotlib inline
```

---

<sup>9</sup> Более подробно все алгоритмы обучения без учителя, поддерживаемые библиотекой Scikit-learn, описаны в официальной документации ([https://scikit-learn.org/stable/unsupervised\\_learning.html](https://scikit-learn.org/stable/unsupervised_learning.html)).

Теперь можно приступить к созданию тестового набора. Результат представлен на рис. 13.24.

```
In [3]: from sklearn.datasets.samples_generator import make_blobs
```

```
In [4]: X, y = make_blobs(n_samples=250, centers=4,  
                           random_state=500, cluster_std=1.25) ❶
```

```
In [5]: plt.figure(figsize=(10, 6))  
        plt.scatter(X[:, 0], X[:, 1], s=50);
```

❶ Создание тестового набора из 250 выборок, кластеризуемых вокруг четырех центров.

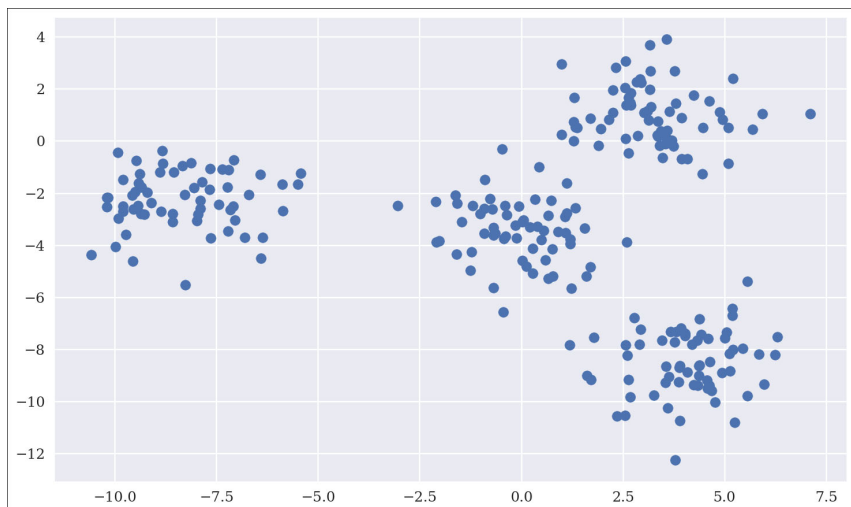


Рис. 13.24. Тестовые данные для алгоритма кластеризации

### Кластеризация методом *k*-средних

Одно из преимуществ библиотеки Scikit-learn — наличие стандартизированного программного интерфейса для применения различных алгоритмов. Приведенный ниже код выполняет базовые действия для кластеризации методом *k*-средних, которые затем повторяются для других моделей:

- импорт класса модели;
- инициализация объекта модели;
- обучение модели на тестовых данных;
- прогнозирование результатов в рамках обученной модели.

Результаты показаны на рис. 13.25.

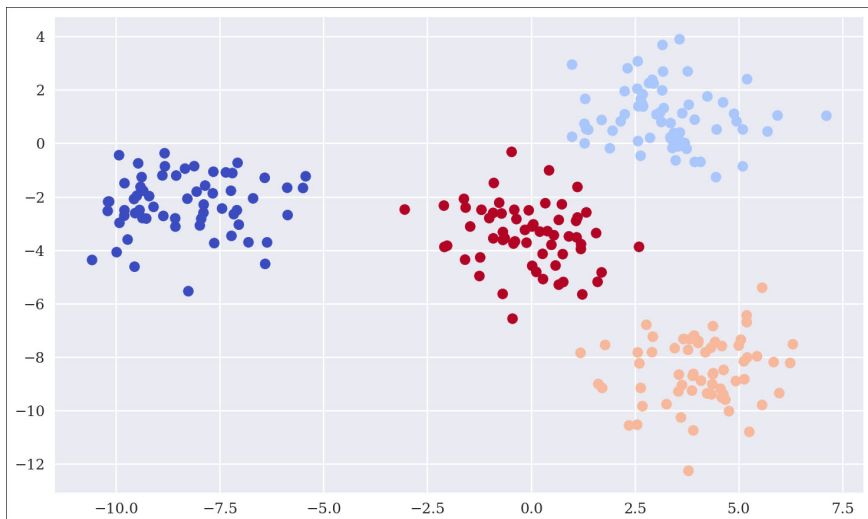


Рис. 13.25. Тестовые данные и распознанные кластеры

```
In [6]: from sklearn.cluster import KMeans ❶
```

```
In [7]: model = KMeans(n_clusters=4, random_state=0) ❷
```

```
In [8]: model.fit(X) ❸
```

```
Out[8]: KMeans(algorithm='auto', copy_x=True, init='k-means++',  
              max_iter=300, n_clusters=4, n_init=10, n_jobs=None,  
              precompute_distances='auto', random_state=0,  
              tol=0.0001, verbose=0)
```

```
In [9]: y_kmeans = model.predict(X) ❹
```

```
In [10]: y_kmeans[:12] ❺
```

```
Out[10]: array([1, 1, 0, 3, 0, 1, 3, 3, 3, 0, 2, 2], dtype=int32)
```

```
In [11]: plt.figure(figsize=(10, 6))  
         plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='coolwarm');
```

- ❶ Импорт класса модели из библиотеки Scikit-learn.
- ❷ Инициализация объекта модели; значения параметров подобраны в соответствии с особенностями тестового набора данных.

- ③ Обучение объекта модели на необработанных данных.
- ④ Прогнозирование (номеров) кластеров в необработанных данных.
- ⑤ Вывод спрогнозированных номеров кластеров.

### Смесь гауссиан

В качестве альтернативы методу кластеризации рассмотрим смесь гауссиан. Порядок действий здесь тот же самый, и при аналогичной параметризации результаты будут теми же.

```
In [12]: from sklearn.mixture import GaussianMixture
```

```
In [13]: model = GaussianMixture(n_components=4, random_state=0)
```

```
In [14]: model.fit(X)
```

```
Out[14]: GaussianMixture(covariance_type='full',  
                           init_params='kmeans',  
                           max_iter=100, means_init=None,  
                           n_components=4, n_init=1,  
                           precisions_init=None, random_state=0,  
                           reg_covar=1e-06, tol=0.001, verbose=0,  
                           verbose_interval=10, warm_start=False,  
                           weights_init=None)
```

```
In [15]: y_gm = model.predict(X)
```

```
In [16]: y_gm[:12]
```

```
Out[16]: array([1, 1, 0, 3, 0, 1, 3, 3, 3, 0, 2, 2])
```

```
In [17]: (y_gm == y_kmeans).all() ❶
```

```
Out[17]: True
```

- ❶ Результаты применения метода  $k$ -средних и смеси гауссиан оказываются одинаковыми.

## Обучение с учителем

Обучение с учителем представляет собой концепцию машинного обучения, согласно которой система получает определенные подсказки в виде известных результатов или наблюдаемых данных. Это означает, что необработанные данные уже содержат то, чему алгоритм машинного обучения должен обучиться.

В последующих примерах мы сконцентрируемся на рассмотрении задач *классификации*, которые противопоставляются задачам *оценивания*. Последние предполагают аппроксимацию реальных значений, тогда как в задачах классификации делается попытка отнести ту или иную комбинацию признаков к определенному классу (представленному целочисленным значением), входящему в сравнительно небольшой набор классов.

В предыдущем разделе мы увидели, что алгоритмы обучения без учителя формируют собственные метки категорий для выявленных кластеров. В случае четырех кластеров метками становятся числа 1, 2, 3 и 4. В обучении с учителем такие метки предоставляются заранее, что позволяет алгоритму обучаться *зависимостям* между отдельными признаками и категориями (классами) данных. Другими словами, на этапе обучения модели алгоритм *уже знает* правильный класс для заданной комбинации признаков.

В этом разделе мы рассмотрим применение следующих алгоритмов классификации: гауссовский наивный байесовский классификатор, логистическая регрессия, деревья принятия решений, глубокие нейронные сети и метод опорных векторов<sup>10</sup>.

## Данные

Библиотека Scikit-learn также позволяет создавать тестовые выборки для алгоритмов классификации. Чтобы иметь возможность визуализировать результаты, тестовый набор должен включать два вещественных информативных признака и бинарную метку (0 или 1). В следующем коде создается тестовый набор, выводится фрагмент набора и строится диаграмма (рис. 13.26).

```
In [18]: from sklearn.datasets import make_classification
```

```
In [19]: n_samples = 100
```

```
In [20]: X, y = make_classification(n_samples=n_samples, n_features=2,  
                                   n_informative=2, n_redundant=0,  
                                   n_repeated=0, random_state=250)
```

```
In [21]: X[:5] ❶
```

---

<sup>10</sup> Более подробно все алгоритмы обучения с учителем, поддерживаемые библиотекой Scikit-learn, описаны в официальной документации ([https://scikit-learn.org/stable/supervised\\_learning.html](https://scikit-learn.org/stable/supervised_learning.html)). Многие из этих алгоритмов применяются не только в задачах классификации, но и в задачах оценивания.

```
Out[21]: array([[ 1.6876, -0.7976],
                [-0.4312, -0.7606],
                [-1.4393, -1.2363],
                [ 1.118 , -1.8682],
                [ 0.0502, 0.659 ]])
```

```
In [22]: X.shape ❶
```

```
Out[22]: (100, 2)
```

```
In [23]: y[:5] ❷
```

```
Out[23]: array([1, 0, 0, 1, 1])
```

```
In [24]: y.shape ❷
```

```
Out[24]: (100,)
```

```
In [25]: plt.figure(figsize=(10, 6))
         plt.scatter(x=X[:, 0], y=X[:, 1], c=y, cmap='coolwarm');
```

❶ Пары вещественных признаков.

❷ Бинарная метка.

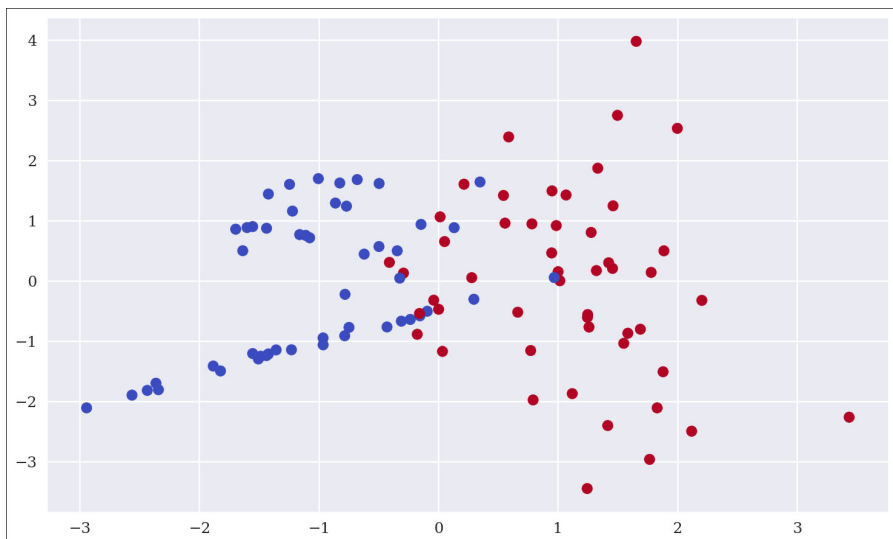


Рис. 13.26. Тестовые данные для алгоритма классификации

## Гауссовский наивный байесовский классификатор

Гауссовский наивный байесовский классификатор (Gaussian Naive Bayes — GNB) считается хорошим базовым алгоритмом для широкого круга задач классификации. Порядок действий здесь такой же, как и в методе  $k$ -средних.

```
In [26]: from sklearn.naive_bayes import GaussianNB
         from sklearn.metrics import accuracy_score
```

```
In [27]: model = GaussianNB()
```

```
In [28]: model.fit(X, y)
```

```
Out[28]: GaussianNB(priors=None, var_smoothing=1e-09)
```

```
In [29]: model.predict_proba(X).round(4)[:5]
```

```
Out[29]: array([[0.0041, 0.9959],
                [0.8534, 0.1466],
                [0.9947, 0.0053],
                [0.0182, 0.9818],
                [0.5156, 0.4844]])
```

```
In [30]: pred = model.predict(X) ②
```

In [31]: pred ②

```
Out[31]: array([1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0,
                1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1,
                0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1,
                0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
                1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0,
                0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0])
```

```
In [32]: pred == y ③
```

```
Out[32]: array([[ True,  True,  True,  True, False,  True,  True,
        True,  True,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,
        True, False, False, False,  True,  True,  True,
        True,  True,  True,  True,  True, False,  True,
        True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,
        True,  True, False,  True, False,  True,  True,
        True,  True,  True,  True,  True,  True,  True,
        True,  True, False,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,
        True, False,  True, False,  True,  True,  True,
```



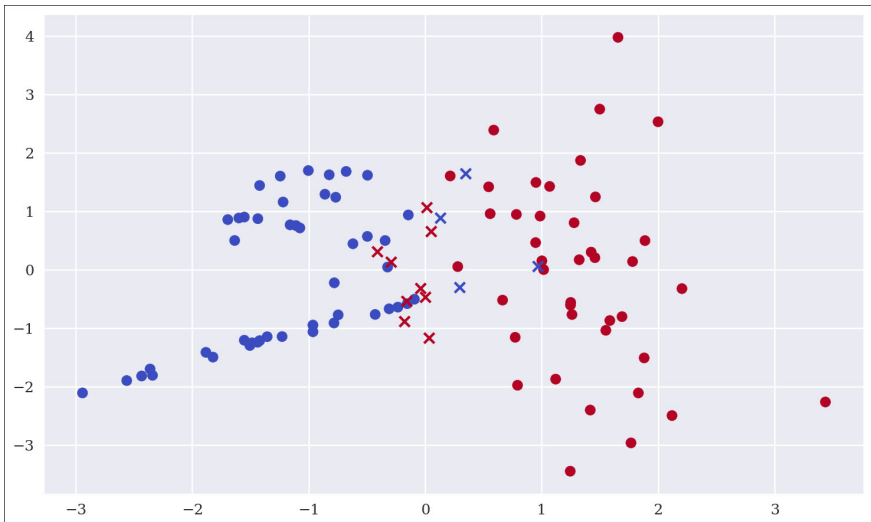
```
True, True, True, True, True, True, True,  
True, False, True, False, True, True, True,  
True, True])
```

```
In [33]: accuracy_score(y, pred) ❹
```

```
Out[33]: 0.87
```

- ❶ Вывод вероятностей, присваиваемых каждому классу после обучения модели.
- ❷ Прогнозирование бинарных классов для набора данных на основе полученных вероятностей.
- ❸ Сравнение предсказанных классов с действительными.
- ❹ Оценка точности прогнозируемых значений.

На рис. 13.27 визуализированы правильные и ложные прогнозы, полученные классификатором GNB.



*Рис. 13.27. Правильные (точки) и ложные (крестики) прогнозы, полученные классификатором GNB*

```
In [34]: Xc = X[y == pred] ❶
```

```
        Xf = X[y != pred] ❷
```

```
In [35]: plt.figure(figsize=(10, 6))
```

```
plt.scatter(x=Xc[:, 0], y=Xc[:, 1], c=y[y == pred],
            marker='o', cmap='coolwarm') ❶
plt.scatter(x=Xf[:, 0], y=Xf[:, 1], c=y[y != pred],
            marker='x', cmap='coolwarm') ❷
```

- ❶ Выбор и отображение *правильных* прогнозов.
- ❷ Выбор и отображение *ложных* прогнозов.

## Логистическая регрессия

*Логистическая регрессия* — это быстрый и хорошо масштабируемый алгоритм классификации. Точность в данном конкретном случае получается не-много выше, чем у классификатора GNB.

```
In [36]: from sklearn.linear_model import LogisticRegression
```

```
In [37]: model = LogisticRegression(C=1, solver='lbfgs')
```

```
In [38]: model.fit(X, y)
```

```
Out[38]: LogisticRegression(C=1, class_weight=None, dual=False,
                             fit_intercept=True, intercept_scaling=1,
                             max_iter=100, multi_class='warn',
                             n_jobs=None, penalty='l2',
                             random_state=None, solver='lbfgs',
                             tol=0.0001, verbose=0, warm_start=False)
```

```
In [39]: model.predict_proba(X).round(4)[:5]
```

```
Out[39]: array([[0.011 , 0.989 ],
                 [0.7266, 0.2734],
                 [0.971 , 0.029 ],
                 [0.04  , 0.96  ],
                 [0.4843, 0.5157]])
```

```
In [40]: pred = model.predict(X)
```

```
In [41]: accuracy_score(y, pred)
```

```
Out[41]: 0.9
```

```
In [42]: Xc = X[y == pred]
         Xf = X[y != pred]
```

```
In [43]: plt.figure(figsize=(10, 6))
         plt.scatter(x=Xc[:, 0], y=Xc[:, 1], c=y[y == pred],
                     marker='o', cmap='coolwarm')
```

```
plt.scatter(x=Xf[:, 0], y=Xf[:, 1], c=y[y != pred],
            marker='x', cmap='coolwarm');
```

## Дерево решений

*Дерево решений* — еще один хорошо масштабируемый алгоритм классификации. При максимальной глубине 1 он уже обеспечивает немного более высокую точность, чем наивный байесовский классификатор и логистическая регрессия (рис. 13.28).

```
In [44]: from sklearn.tree import DecisionTreeClassifier
```

```
In [45]: model = DecisionTreeClassifier(max_depth=1)
```

```
In [46]: model.fit(X, y)
```

```
Out[46]: DecisionTreeClassifier(class_weight=None, criterion='gini',
                                max_depth=1, max_features=None,
                                max_leaf_nodes=None,
                                min_impurity_decrease=0.0,
                                min_impurity_split=None,
                                min_samples_leaf=1,
                                min_samples_split=2,
                                min_weight_fraction_leaf=0.0,
                                presort=False, random_state=None,
                                splitter='best')
```

```
In [47]: model.predict_proba(X).round(4)[:5]
```

```
Out[47]: array([[0.08, 0.92],
                [0.92, 0.08],
                [0.92, 0.08],
                [0.08, 0.92],
                [0.08, 0.92]])
```

```
In [48]: pred = model.predict(X)
```

```
In [49]: accuracy_score(y, pred)
```

```
Out[49]: 0.92
```

```
In [50]: Xc = X[y == pred]
         Xf = X[y != pred]
```

```
In [51]: plt.figure(figsize=(10, 6))
```

```
plt.scatter(x=Xc[:, 0], y=Xc[:, 1], c=y[y == pred],
            marker='o', cmap='coolwarm')
plt.scatter(x=Xf[:, 0], y=Xf[:, 1], c=y[y != pred],
            marker='x', cmap='coolwarm');
```

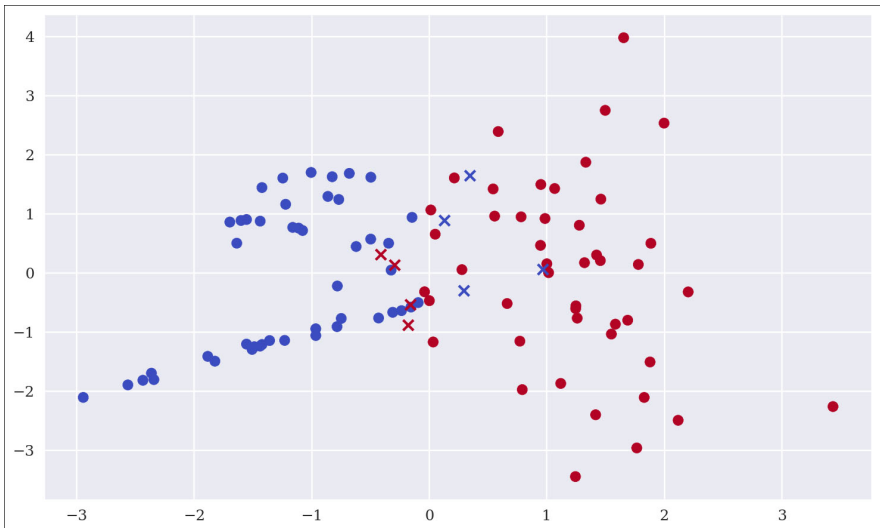


Рис. 13.28. Правильные (точки) и ложные (крестики) прогнозы, полученные с помощью дерева решений ( $\text{max\_depth}=1$ )

Если увеличить максимальную глубину дерева решений, то можно получить идеальные результаты.

```
In [52]: print('{:>8s} | {:>8s}'.format('Глубина', 'Точность'))
          print(20 * '-')
          for depth in range(1, 7):
              model = DecisionTreeClassifier(max_depth=depth)
              model.fit(X, y)
              acc = accuracy_score(y, model.predict(X))
              print('{:>8d} | {:>8.2f}'.format(depth, acc))
          Глубина | Точность
```

```
-----
          1 |      0.92
          2 |      0.92
          3 |      0.94
          4 |      0.97
          5 |      0.99
          6 |      1.00
```

## Глубокие нейронные сети

Глубокие нейронные сети (Deep Neural Networks — DNN) считаются одним из самых мощных, хотя и вычислительно затратных, алгоритмов прогнозирования и классификации. Своей популярностью они во многом обязаны открытому пакету TensorFlow компании Google и другим технологическим достижениям последних лет. DNN способны обучаться и моделировать сложные нелинейные взаимосвязи. Несмотря на то что нейронные сети появились еще в 1970-е годы, их полномасштабная практическая реализация стала возможной только с появлением высокопроизводительных процессоров (CPU, GPU, TPU), эффективных численных алгоритмов и соответствующего программного обеспечения.

Другие алгоритмы машинного обучения, в частности линейные модели логистической регрессии, способны эффективно обучаться при решении стандартных задач оптимизации, однако DNN основаны на *глубоком обучении*, что требует многократного повторения одних и тех же этапов настройки определенных параметров (весов) и сравнения полученных результатов с исходными данными. В этом смысле глубокое обучение можно сравнить с применением метода Монте-Карло в финансовой математике, когда, к примеру, для оценки европейского колл-опциона необходимо смоделировать 100 000 траекторий базового актива. С другой стороны, формула оценки опционов Блэка — Шоулза — Мертона существует в замкнутой форме и может быть вычислена аналитическим способом.

Высокая гибкость и вычислительная эффективность метода Монте-Карло достигается за счет повышенных требований к оборудованию и оперативной памяти. То же самое справедливо и в отношении глубокого обучения, которое намного гибче других алгоритмов машинного обучения, но для этого требуется больше вычислительных ресурсов.

### DNN и библиотека Scikit-learn

Несмотря на иное предназначение, библиотека Scikit-learn реализует такой же программный интерфейс для класса алгоритмов `MLPClassifier`<sup>11</sup> (модель DNN, классификатор многослойного перцептрона), как и для других алгоритмов машинного обучения. Располагая всего двумя так называемыми *скрытыми слоями*, эта модель позволяет достичь оптимального результата на тестовых данных. (Скрытые слои — то, что отличает глубокое обучение от простого машинного обучения. В этой модели веса “обучаются”, например в контексте

---

<sup>11</sup> Подробное описание класса доступно в документации ([https://scikit-learn.org/stable/modules/generated/sklearn.neural\\_network.MLPClassifier.html](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html)).

линейной регрессии, а не вычисляются напрямую с помощью регрессии по методу наименьших квадратов.)

```
In [53]: from sklearn.neural_network import MLPClassifier
```

```
In [54]: model = MLPClassifier(solver='lbfgs', alpha=1e-5,  
                               hidden_layer_sizes=2 * [75],  
                               random_state=10)
```

```
In [55]: %time model.fit(X, y)  
CPU times: user 537 ms, sys: 14.2 ms, total: 551 ms  
Wall time: 340 ms
```

```
Out[55]: MLPClassifier(activation='relu', alpha=1e-05,  
                        batch_size='auto', beta_1=0.9,  
                        beta_2=0.999, early_stopping=False,  
                        epsilon=1e-08, hidden_layer_sizes=[75, 75],  
                        learning_rate='constant',  
                        learning_rate_init=0.001, max_iter=200,  
                        momentum=0.9, n_iter_no_change=10,  
                        nesterovs_momentum=True, power_t=0.5,  
                        random_state=10, shuffle=True,  
                        solver='lbfgs', tol=0.0001,  
                        validation_fraction=0.1, verbose=False,  
                        warm_start=False)
```

```
In [56]: pred = model.predict(X)  
pred
```

```
Out[56]: array([1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0,  
                1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0,  
                0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1,  
                1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0,  
                1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0,  
                0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0])
```

```
In [57]: accuracy_score(y, pred)
```

```
Out[57]: 1.0
```

## DNN и библиотека TensorFlow

Программный интерфейс библиотеки TensorFlow отличается от стандарта, принятого в Scikit-learn. Тем не менее схема применения класса `DNNClassifier` такая же простая.

```
In [58]: import tensorflow as tf  
         tf.logging.set_verbosity(tf.logging.ERROR) ❶
```

```

In [59]: fc = [tf.contrib.layers.real_valued_column('features')] ❷

In [60]: model = tf.contrib.learn.DNNClassifier(hidden_units=5 * [250],
                                                n_classes=2,
                                                feature_columns=fc) ❸

In [61]: def input_fn(): ❹
        fc = {'features': tf.constant(X)}
        la = tf.constant(y)
        return fc, la

In [62]: %time model.fit(input_fn=input_fn, steps=100) ❺
CPU times: user 7.1 s, sys: 1.35 s, total: 8.45 s
Wall time: 4.71 s
Out[62]: DNNClassifier(params={'head': <tensorflow.contrib.learn.
python.learn ... head._BinaryLogisticHead object at
0x1a3ee692b0>, 'hidden_units': [250, 250, 250, 250, 250],
'feature_columns': (_RealValuedColumn(column_name='features',
dimension=1, default_value=None, dtype=tf.float32,
normalizer=None)), 'optimizer': None, 'activation_fn':
<function relu at 0x1a3aa75b70>, 'dropout': None,
'gradient_clip_norm': None, 'embedding_lr_multipliers': None,
'input_layer_min_slice_size': None})

In [63]: model.evaluate(input_fn=input_fn, steps=1) ❻
Out[63]: {'loss': 0.18724777,
'accuracy': 0.91,
'labels/prediction_mean': 0.5003989,
'labels/actual_label_mean': 0.5,
'accuracy/baseline_label_mean': 0.5,
'auc': 0.9782,
'auc_precision_recall': 0.97817385,
'accuracy/threshold_0.500000_mean': 0.91,
'precision/positive_threshold_0.500000_mean': 0.9019608,
'recall/positive_threshold_0.500000_mean': 0.92,
'global_step': 100}

In [64]: pred = np.array(list(model.predict(input_fn=input_fn))) ❼
pred[:10] ❺
Out[64]: array([1, 0, 0, 1, 1, 0, 1, 1, 1, 1])

In [65]: %time model.fit(input_fn=input_fn, steps=750) ❻

```

```

CPU times: user 29.8 s, sys: 7.51 s, total: 37.3 s
Wall time: 13.6 s
Out[65]: DNNClassifier(params={'head': <tensorflow.contrib.learn.
python.learn ... head._BinaryLogisticHead object at
0x1a3ee692b0>, 'hidden_units': [250, 250, 250, 250, 250],
'feature_columns': (_RealValuedColumn(column_name=
'features', dimension=1, default_value=None, dtype=
tf.float32, normalizer=None)), 'optimizer': None,
'activation_fn': <function relu at 0x1a3aa75b70>,
'dropout': None, 'gradient_clip_norm': None,
'embedding_lr_multipliers': None,
'input_layer_min_slice_size': None})

In [66]: model.evaluate(input_fn=input_fn, steps=1) ❸
Out[66]: {'loss': 0.09271307,
'accuracy': 0.94,
'labels/prediction_mean': 0.5274486,
'labels/actual_label_mean': 0.5,
'accuracy/baseline_label_mean': 0.5,
'auc': 0.99759996,
'auc_precision_recall': 0.9977609,
'accuracy/threshold_0.500000_mean': 0.94,
'precision/positive_threshold_0.500000_mean': 0.9074074,
'recall/positive_threshold_0.500000_mean': 0.98,
'global_step': 850}

```

- ❶ Задание уровня детализации для журнала TensorFlow.
- ❷ Абстрактное определение вещественных признаков.
- ❸ Создание объекта модели.
- ❹ Данные признаков и меток вводятся с помощью функции.
- ❺ Обучение и оценка модели.
- ❻ Прогнозирование меток на основе признаков.
- ❼ Повторное обучение модели с увеличением числа шагов; предыдущие результаты служат отправной точкой.
- ❽ В результате повторного обучения точность модели повысилась.

Этот пример показывает лишь малую толику возможностей библиотеки TensorFlow, которая применяется во множестве ответственных проектов, как, например, инициатива компании Alphabet по разработке беспилотных



автомобилей. Что касается скорости обучения моделей TensorFlow, то основной выигрыш достигается за счет использования специализированного оборудования (в первую очередь GPU и TPU вместо CPU).

### ***Преобразование признаков***

Преобразование вещественных признаков может требоваться по целому ряду причин. В следующем примере показаны типичные варианты преобразований. Результаты сравниваются на рис. 13.29.

```
In [67]: from sklearn import preprocessing
```

```
In [68]: X[:5]
```

```
Out[68]: array([[ 1.6876, -0.7976],
                [-0.4312, -0.7606],
                [-1.4393, -1.2363],
                [ 1.118 , -1.8682],
                [ 0.0502,  0.659 ]])
```

```
In [69]: Xs = preprocessing.StandardScaler().fit_transform(X) ❶
        Xs[:5]
```

```
Out[69]: array([[ 1.2881, -0.5489],
                [-0.3384, -0.5216],
                [-1.1122, -0.873 ],
                [ 0.8509, -1.3399],
                [ 0.0312,  0.5273]])
```

```
In [70]: Xm = preprocessing.MinMaxScaler().fit_transform(X) ❷
        Xm[:5]
```

```
Out[70]: array([[0.7262, 0.3563],
                [0.3939, 0.3613],
                [0.2358, 0.2973],
                [0.6369, 0.2122],
                [0.4694, 0.5523]])
```

```
In [71]: Xn1 = preprocessing.Normalizer(norm='l1').transform(X) ❸
        Xn1[:5]
```

```
Out[71]: array([[ 0.6791, -0.3209],
                [-0.3618, -0.6382],
                [-0.5379, -0.4621],
                [ 0.3744, -0.6256],
                [ 0.0708,  0.9292]])
```

```
In [72]: Xn2 = preprocessing.Normalizer(norm='l2').transform(X) ❹
```

```

Xn2[:5]
Out[72]: array([[ 0.9041, -0.4273],
                [-0.4932, -0.8699],
                [-0.7586, -0.6516],
                [ 0.5135, -0.8581],
                [ 0.076 ,  0.9971]])

In [73]: plt.figure(figsize=(10, 6))
markers = ['o', '.', 'x', '^', 'v']
data_sets = [X, Xs, Xm, Xn1, Xn2]
labels = ['Необработанные данные', 'Стандартное норм.
          распределение', 'Минимакс', 'Норма L1', 'Норма L2']
for x, m, l in zip(data_sets, markers, labels):
    plt.scatter(x=x[:, 0], y=x[:, 1], c=y,
               marker=m, cmap='coolwarm', label=l)
plt.legend();

```

- ❶ Приведение признаков к стандартному нормальному распределению с нулевым средним и единичной дисперсией.
- ❷ Приведение каждого признака к требуемому диапазону, определяемому минимальным и максимальным значениями.
- ❸ Индивидуальное масштабирование признаков в соответствии с вектор-нормой L1 или L2.

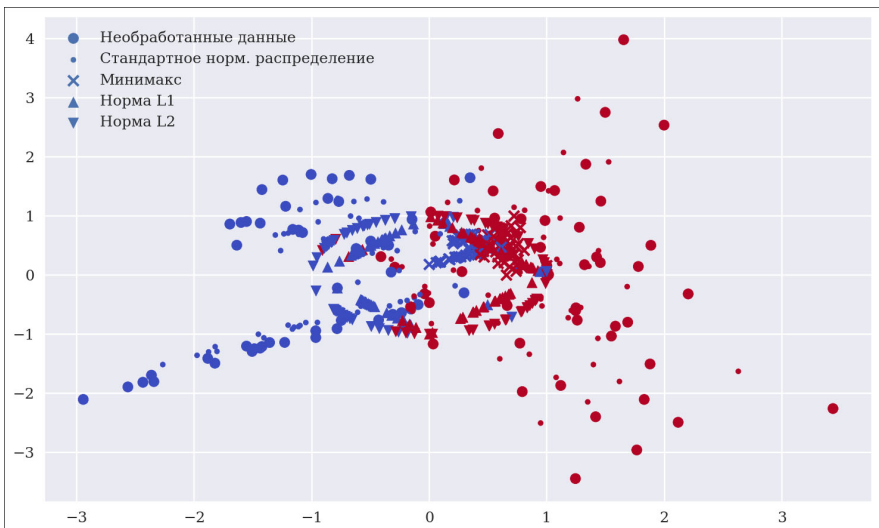


Рис. 13.29. Сравнение исходных и преобразованных данных

В задачах распознавания образов преобразование данных в формат категориальных признаков часто помогает получить приемлемые результаты. Для этого вещественные значения признаков транслируются в ограниченное, фиксированное количество возможных целочисленных значений (категорий, классов).

```
In [74]: X[:5]
```

```
Out[74]: array([[ 1.6876, -0.7976],
                [-0.4312, -0.7606],
                [-1.4393, -1.2363],
                [ 1.118 , -1.8682],
                [ 0.0502,  0.659 ]])
```

```
In [75]: Xb = preprocessing.Binarizer().fit_transform(X) ❶
        Xb[:5]
```

```
Out[75]: array([[1., 0.],
                [0., 0.],
                [0., 0.],
                [1., 0.],
                [1., 1.]])
```

```
In [76]: 2 ** 2 ❷
```

```
Out[76]: 4
```

```
In [77]: Xd = np.digitize(X, bins=[-1, 0, 1]) ❸
        Xd[:5]
```

```
Out[77]: array([[3, 1],
                [1, 1],
                [0, 0],
                [3, 0],
                [2, 2]])
```

```
In [78]: 4 ** 2 ❹
```

```
Out[78]: 16
```

- ❶ Преобразование признаков в бинарные категории.
- ❷ Количество возможных комбинаций значений у двух бинарных признаков.
- ❸ Преобразование признаков в категориальные признаки на основе списка значений, применяемых при сортировке.
- ❹ Количество возможных комбинаций значений при сортировке двух признаков по трем значениям.

## Разделение обучающих и тестовых наборов: метод опорных векторов

На данном этапе у любого опытного разработчика систем машинного обучения наверняка возникли вопросы по поводу приведенных реализаций: во всех примерах одни и те же данные использовались как для обучения, так и для прогнозирования. Конечно же, работу алгоритма машинного обучения лучше всего оценивать, используя для обучения и тестирования разные (под)наборы данных. Это приближает нас к реальным сценариям.

В библиотеке Scikit-learn имеется удобная функция `train_test_split()`, которая позволяет случайным (но воспроизводимым) образом разделять наборы данных на обучающие (тренировочные) и тестовые. В следующем примере применяется еще один алгоритм классификации: *метод опорных векторов* (Support Vector Machine — SVM). Сначала модель обучается на тренировочном наборе.

```
In [79]: from sklearn.svm import SVC
         from sklearn.model_selection import train_test_split

In [80]: train_x, test_x, train_y, test_y = train_test_split(X, y,
                                                            test_size=0.33, random_state=0)

In [81]: model = SVC(C=1, kernel='linear')

In [82]: model.fit(train_x, train_y) ❶
Out[82]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovr', degree=3,
            gamma='auto_deprecated', kernel='linear', max_iter=-1,
            probability=False, random_state=None, shrinking=True,
            tol=0.001, verbose=False)

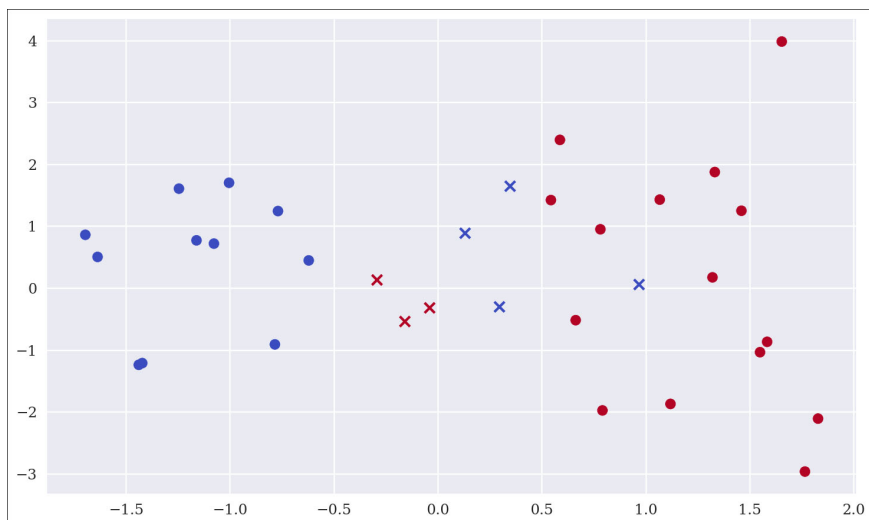
In [83]: pred_train = model.predict(train_x) ❷

In [84]: accuracy_score(train_y, pred_train) ❸
Out[84]: 0.9402985074626866
```

- ❶ Обучение модели на тренировочном наборе.
- ❷ Прогнозирование меток для тренировочного набора.
- ❸ Точность прогнозирования для тренировочного набора (в пределах выборки).

Далее мы проверяем обученную модель на тестовом наборе. На рис. 13.30 показаны правильные и ложные прогнозы для тестовых данных. Как и следо-

вало ожидать, точность модели на тестовых данных оказалась ниже, чем на тренировочных данных.



*Рис. 13.30. Правильные (точки) и ложные (крестики) прогнозы, полученные с помощью SVM-модели для тестовых данных*

```
In [85]: pred_test = model.predict(test_x) ❶
```

```
In [86]: test_y == pred_test ❷
```

```
Out[86]: array([ True,  True,  True,  True,  True,  True,  True,
        True,  True,  True, False, False, False,  True,
        True,  True, False, False, False,  True,  True,
        True,  True,  True,  True,  True,  True,  True,
        True,  True,  True, False,  True])
```

```
In [87]: accuracy_score(test_y, pred_test) ❷
```

```
Out[87]: 0.7878787878787878
```

```
In [88]: test_c = test_x[test_y == pred_test]
        test_f = test_x[test_y != pred_test]
```

```
In [89]: plt.figure(figsize=(10, 6))
        plt.scatter(x=test_c[:, 0], y=test_c[:, 1], c=test_y[
            test_y == pred_test], marker='o',
            cmap='coolwarm')
```

```
plt.scatter(x=test_f[:, 0], y=test_f[:, 1], c=test_y[
    test_y != pred_test], marker='x',
    cmap='coolwarm');
```

- ❶ Прогнозирование меток для тестового набора.
- ❷ Точность обученной модели для тестового набора (вне выборки).

SVM-классификатор поддерживает несколько вариантов используемых ядер. При этом, как показывает следующий пример, разные ядра могут приводить к совершенно разным результатам (т.е. обеспечивать разную точность), в зависимости от решаемой задачи. В примере сначала выполняется преобразование вещественных признаков в категориальные.

```
In [90]: bins = np.linspace(-4.5, 4.5, 50)
```

```
In [91]: Xd = np.digitize(X, bins=bins)
```

```
In [92]: Xd[:5]
```

```
Out[92]: array([[34, 21],
               [23, 21],
               [17, 18],
               [31, 15],
               [25, 29]])
```

```
In [93]: train_x, test_x, train_y, test_y = train_test_split(Xd, y,
    test_size=0.33, random_state=0)
```

```
In [94]: print('{:>8s} | {:>8s}'.format('Ядро', 'Точность'))
print(20 * '-')
for kernel in ['linear', 'poly', 'rbf', 'sigmoid']:
    model = SVC(C=1, kernel=kernel, gamma='auto')
    model.fit(train_x, train_y)
    acc = accuracy_score(test_y, model.predict(test_x))
    print('{:>8s} | {:8.3f}'.format(kernel, acc))
    Ядро | Точность
```

```
-----
linear |    0.848
poly   |    0.758
rbf    |    0.788
sigmoid |    0.455
```

## Резюме

Статистика — важнейшая дисциплина, которая широко применяется для решения всевозможных задач в самых разных отраслях, включая финансы и общественные науки. В одной главе невозможно охватить столь широкую область знаний. Поэтому мы сфокусировались на четырех ключевых темах, рассмотрев применение Python и нескольких научных библиотек для решения прикладных задач.

### *Нормальное распределение*

Предположение о нормальном распределении значений доходности ценных бумаг лежит в основе многих финансовых теорий и приложений, поэтому важно уметь проверять, справедлива ли эта гипотеза для конкретного временного ряда. Как было показано, реальные данные о доходности обычно *не* соответствуют нормальному распределению.

### *Оптимизация портфеля*

Портфельная теория Марковица, делающая акцент на оценке среднего значения и дисперсии/волатильности доходности, стала одним из самых первых и самых главных достижений в области финансовой статистики. С ее помощью удобно рассматривать концепцию *диверсификации* инвестиций.

### *Байесовская статистика*

Байесовская статистика в целом (и байесовская регрессия в частности) стала важным инструментом финансовых вычислений, так как позволяет преодолеть недостатки многих подходов, в том числе тех, которые рассматривались в главе 11. Ее математический аппарат может показаться достаточно сложным, но основные идеи (такие, как изменение степени доверия со временем) воспринимаются достаточно легко (по крайней мере, на интуитивном уровне).

### *Машинное обучение*

В наши дни машинное обучение применяется в финансовой среде наряду с традиционными статистическими методами. В этой главе мы изучили алгоритмы обучения без учителя (такие, как кластеризация методом *k*-средних) и с учителем (например, классификаторы глубоких нейронных сетей), а также рассмотрели ряд дополнительных тем, таких как преобразование признаков и разделение наборов данных на обучающие (тренировочные) и тестовые.

## Дополнительные ресурсы

Дополнительную информацию о рассмотренных в этой главе темах и пакетах можно найти в Интернете:

- документация по статистическим функциям SciPy (<https://docs.scipy.org/doc/scipy/reference/stats.html>);
- документация к библиотеке statsmodels (<http://www.statsmodels.org/stable/>);
- детальное описание функций оптимизации (<https://docs.scipy.org/doc/scipy/reference/optimize.html>);
- документация к пакету PyMC3 (<https://docs.pymc.io/>);
- документация к библиотеке Scikit-learn (<https://scikit-learn.org/>).

В главе упоминались следующие источники.

1. Albon, Chris. *Machine Learning with Python Cookbook* (2018, O'Reilly).
2. Alpaydin, Ethem. *Machine Learning* (2016, MIT Press).
3. Copeland, Thomas, Fred Weston, and Kuldeep Shastri. *Financial Theory and Corporate Policy* (2005, Pearson).
4. Downey, Allen. *Think Bayes* (2013, O'Reilly).
5. Geweke, John. *Contemporary Bayesian Econometrics and Statistics* (2005, Wiley).
6. Glasserman, Paul. *Monte Carlo Methods in Financial Engineering* (2004, Springer).
7. James, Gareth, et al. *An Introduction to Statistical Learning — With Applications in R* (2013, Springer).
8. Lopez de Prado, Marcos. *Advances in Financial Machine Learning* (2018, Wiley).
9. Markowitz, Harry. *Portfolio Selection* (1952, *Journal of Finance*, Vol. 7, pp. 77–91).
10. Rachev, Svetlozar, et al. *Bayesian Methods in Finance* (2008, Wiley).
11. VanderPlas, Jake. *Python Data Science Handbook* (2016, O'Reilly).





---

# Алгоритмическая торговля

В этой части речь пойдет о применении Python для алгоритмической торговли. Все больше торговых платформ и брокеров позволяют своим клиентам, например, использовать REST-совместимые программные интерфейсы для получения исторических или потоковых данных, а также для размещения биржевых заявок. То, что долгое время считалось прерогативой крупных финансовых учреждений, теперь стало доступным даже розничным алгоритмическим трейдерам. В этой среде Python занял доминирующее положение не только как язык программирования, но и как технологическая платформа. Среди прочих факторов это обусловлено в первую очередь тем, что многие торговые площадки, включая FXCM (Forex Capital Markets), предлагают удобные оболочечные пакеты Python для своих REST-совместимых интерфейсов.

Данная часть включает три главы.

- **Глава 14** посвящена знакомству с торговой платформой FXCM, ее программным интерфейсом и оболочечным пакетом `fxcmru`.
- **Глава 15** посвящена применению методов статистического анализа и машинного обучения для разработки стратегий алгоритмической торговли. Будет также показано, как выполнять векторизованное тестирование на исторических данных.
- **Глава 16** посвящена внедрению автоматизированных стратегий алгоритмической торговли. Речь пойдет об управлении капиталом, тестировании операций на производительность и рисковость, онлайн-алгоритмах и развертывании кода.



# Торговая платформа FXCM

Финансовым учреждениям нравится называть то, что они делают, торговлей. Давайте будем честными: это не торговля, а ставки.

*Грейдон Картер*

В этой главе мы познакомимся с торговой платформой FXCM (Forex Capital Markets), ее программным интерфейсом, поддерживающим потоковую обработку данных, и оболочечным пакетом *fxстру*. FXCM предлагает розничным и корпоративным трейдерам ряд финансовых продуктов, которые могут торговаться как через традиционные приложения, так и с помощью специализированного программного интерфейса. Основной акцент в этих продуктах делается на валютных парах, а также *контрактах на разницу цен*, или CFD (Contract For Difference), на основные фондовые индексы, биржевые товары и т.п.



## Предупреждение о высоких рисках

Работа на рынках форекс/CFD связана с высокими рисками и подходит далеко не всем инвесторам, поскольку они рискуют своими залогами депозитами. Кредитное плечо может работать против них. Рассматриваемые продукты предназначены для профессиональных брокеров. Прежде чем заниматься маржинальной торговлей, убедитесь, что в полной мере осознаете все риски и последствия принимаемых решений. Старайтесь трезво оценивать свои финансовые возможности и уровень квалификации. Какая бы информация к вам ни поступала (экспертные мнения, новости рынка, данные исследований, аналитические отчеты, ценовые сводки и т.п.), ее нужно лишь принимать к сведению, но не рассматривать как призыв к инвестиционным действиям. Любые комментарии делаются участниками рынка в частном порядке, без соблюдения юридических процедур, гарантирующих независимость инвестиционных решений, и поэтому не подвергаются никакой цензуре. Ни платформа FXCM, ни автор книги не несут ответственности за возможные убытки или финансовый ущерб от (прямого или косвенного) влияния такого рода информации на принятие инвестором окончательных решений.

Платформа FXCM позволяет внедрять стратегии алгоритмической торговли даже частным трейдерам с небольшим капиталом.

В этой главе рассматриваются основные функциональные возможности программного интерфейса FXCM и пакета `fxcmru`, с помощью которых реализуются автоматизированные стратегии.

### *Настройка программного интерфейса FXCM*

В этом разделе объясняется, как сконфигурировать среду для работы с REST-совместимым программным интерфейсом алгоритмической торговли FXCM.

### *Получение данных*

В этом разделе будет показано, как получать и обрабатывать финансовые данные (в том числе тиковые).

### *Работа с программным интерфейсом FXCM*

В этом разделе рассматриваются типичные задачи, реализуемые с помощью программного интерфейса FXCM, такие как извлечение исторических и потоковых данных, размещение заявок и поиск учетной информации.

## Настройка программного интерфейса FXCM

Подробная документация к программному интерфейсу FXCM доступна по адресу <https://fxcm.github.io/rest-apidocs>. Для установки оболочечного пакета `fxcmru` выполните следующую команду в окне консоли:

```
pip install fxcmru
```

Документация к пакету `fxcmru` доступна на официальной странице (<http://fxcmru.tpq.io>).

Чтобы начать работу с программным интерфейсом FXCM, достаточно создать бесплатную демонстрационную учетную запись (<https://www.fxcm.com/uk/forex-trading-demo/>)<sup>1</sup>. Следующий шаг — создание для учетной записи уникального API-токена, скажем `YOUR_FXCM_API_TOKEN`. Через этот токен осуществляется подключение к серверу FXCM, например так, как показано ниже.

---

<sup>1</sup> Эта возможность доступна не во всех странах. В настоящее время бесплатные учетные записи могут, в частности, создавать жители Армении, Азербайджана, Грузии, Казахстана, Киргизии, Молдавии, Таджикистана, Туркмении и Узбекистана. — *Примеч. ред.*

```
import fxcmru
api = fxcmru.fxcmru(access_token=YOUR_FXCM_API_TOKEN,
                    log_level='error')
```

Альтернативный способ заключается в использовании конфигурационного файла (допустим, *fxcm.cfg*). Содержимое файла должно выглядеть следующим образом.

```
[FXCM]
log_level = error
log_file = PATH_TO_AND_NAME_OF_LOG_FILE
access_token = YOUR_FXCM_API_TOKEN
```

В таком случае для подключения к серверу FXCM нужно ввести следующий код.

```
import fxcmru
api = fxcmru.fxcmru(config_file='fxcm.cfg')
```

По умолчанию класс *fxcmru* подключается к демо-серверу. Но с помощью параметра *server* можно установить соединение и с действующим торговым сервером (при наличии соответствующей учетной записи).

```
api = fxcmru.fxcmru(config_file='fxcm.cfg', server='demo') ❶
```

```
api = fxcmru.fxcmru(config_file='fxcm.cfg', server='real') ❷
```

- ❶ Подключение к демо-серверу.
- ❷ Подключение к действующему торговому серверу.

## Получение данных

FXCM предоставляет доступ к историческим наборам данных о рыночной стоимости акций (содержащим, в частности, тиковые данные) в предварительно упакованном виде. Это означает, что с серверов FXCM можно загрузить сжатые файлы с тиковыми данными, в которых указывается валютный курс EUR/USD, например, за 26-ю неделю 2018 года. Далее будет также показано, как извлекать исторические свечные данные.

## Получение тиковых данных

FXCM дает возможность загружать исторические тиковые данные для большого количества валютных пар. Пакет `fxcm` упрощает получение и последующую обработку таких данных. Для начала импортируем все необходимые модули.

```
In [1]: import time
        import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt
```

```
In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

Теперь посмотрим доступные символы (валютные пары), для которых имеются тиковые данные.

```
In [3]: from fxcm.py import fxcm.py_tick_data_reader as tdr
```

```
In [4]: print(tdr.get_available_symbols())
('AUDCAD', 'AUDCHF', 'AUDJPY', 'AUDNZD', 'CADCHF', 'EURAUD',
 'EURCHF', 'EURGBP', 'EURJPY', 'EURUSD', 'GBPCHF', 'GBPJPY',
 'GBPNZD', 'GBPUSD', 'GBPCHF', 'GBPJPY', 'GBPNZD', 'NZDCAD',
 'NZDCHF', 'NZDJPY', 'NZDUSD', 'USDCAD', 'USDCHF', 'USDJPY')
```

Ниже показано, как извлечь недельный массив тиковых данных для отдельной пары. Результирующий объект `DataFrame` библиотеки `pandas` содержит более 1,5 млн строк данных.

```
In [5]: start = dt.datetime(2018, 6, 25) ❶
        stop = dt.datetime(2018, 6, 30) ❶
```

```
In [6]: td = tdr('EURUSD', start, stop) ❶
```

```
In [7]: td.get_raw_data().info() ❷
<class 'pandas.core.frame.DataFrame'>
Index: 1963779 entries, 06/24/2018 21:00:12.290 to
      06/29/2018 20:59:00.607
Data columns (total 2 columns):
Bid    float64
Ask    float64
dtypes: float64(2)
memory usage: 44.9+ MB
```

```
In [8]: td.get_data().info() ❸
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1963779 entries, 2018-06-24 21:00:12.290000 to
                                         2018-06-29 20:59:00.607000
Data columns (total 2 columns):
Bid      float64
Ask      float64
dtypes: float64(2)
memory usage: 44.9 MB
```

```
In [9]: td.get_data().head()
Out[9]:
```

	<b>Bid</b>	<b>Ask</b>
<b>2018-06-24 21:00:12.290</b>	1.1662	1.16660
<b>2018-06-24 21:00:16.046</b>	1.1662	1.16650
<b>2018-06-24 21:00:22.846</b>	1.1662	1.16658
<b>2018-06-24 21:00:22.907</b>	1.1662	1.16660
<b>2018-06-24 21:00:23.441</b>	1.1662	1.16663

- ❶ Получение и распаковка файла данных с последующим сохранением необработанных данных в объекте `DataFrame` (в виде атрибута результирующего объекта).
- ❷ Метод `td.get_raw_data()` возвращает объект `DataFrame`, содержащий необработанные данные (индексируются строковыми объектами).
- ❸ Метод `td.get_data()` возвращает объект `DataFrame`, в котором данные индексируются с помощью объектов `DatetimeIndex`.

Благодаря тому, что тиковые данные хранятся в объекте `DataFrame`, можно легко извлекать необходимые поднаборы и выполнять над ними любые аналитические операции. На рис. 14.1 приведены графики среднего спреда и простого скользящего среднего (Simple Moving Average — SMA) отдельного поднабора.

```
In [10]: sub = td.get_data(start='2018-06-29 12:00:00',
                             end='2018-06-29 12:15:00') ❶
```

```
In [11]: sub.head()
Out[11]:
```

	<b>Bid</b>	<b>Ask</b>
<b>2018-06-29 12:00:00.011</b>	1.16497	1.16498
<b>2018-06-29 12:00:00.071</b>	1.16497	1.16497
<b>2018-06-29 12:00:00.079</b>	1.16497	1.16498
<b>2018-06-29 12:00:00.091</b>	1.16495	1.16498
<b>2018-06-29 12:00:00.205</b>	1.16496	1.16498

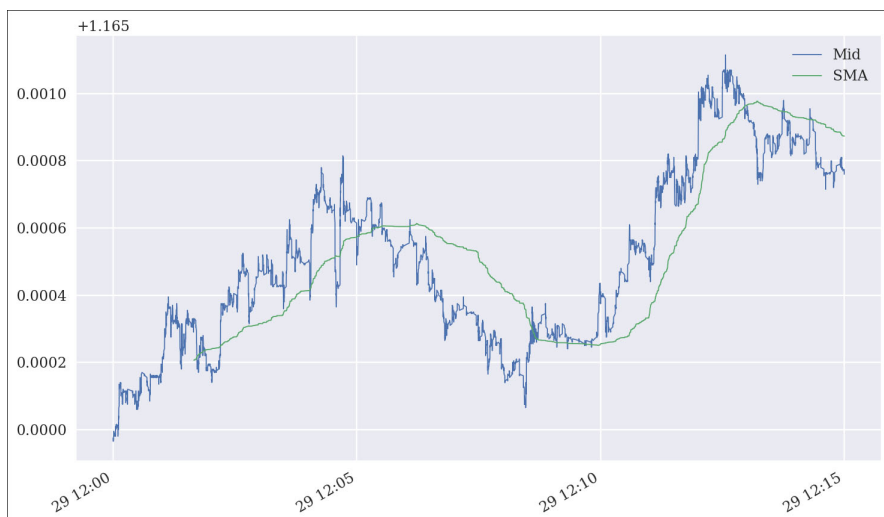
```
In [12]: sub['Mid'] = sub.mean(axis=1) ❷
```



```
In [13]: sub['SMA'] = sub['Mid'].rolling(1000).mean() ❸
```

```
In [14]: sub[['Mid', 'SMA']].plot(figsize=(10, 6), lw=0.75);
```

- ❶ Выбор поднабора из полного набора данных.
- ❷ Вычисление средней разницы между спросом и предложением.
- ❸ Вычисление скользящего среднего (SMA) за интервал, равный 1000 тикам.



*Рис. 14.1. Исторические тиковые средние спреды и скользящее среднее для валютной пары EUR/USD*

## Получение свечных данных

FXCM также предоставляет доступ к историческим свечным данным, т.е. данным за определенные равные интервалы времени. Каждая “свеча” включает цены открытия/закрытия, а также минимума/максимума по спросу и предложению.

Сначала посмотрим доступные символы (валютные пары), для которых имеются свечные данные.

```
In [15]: from fxcmpy import fxcmpy_candles_data_reader as cdr
```

```
In [16]: print(cdr.get_available_symbols())  
( 'AUDCAD', 'AUDCHF', 'AUDJPY', 'AUDNZD', 'CADCHF',  
  'EURAUD', 'EURCHF', 'EURGBP', 'EURJPY', 'EURUSD',
```

```
'GBPCHF', 'GBPJPY', 'GBPUSD', 'GBPCHF',
'GBPJPY', 'GBPUSD', 'NZDCAD', 'NZDCHF', 'NZDJPY',
'NZDUSD', 'USDCAD', 'USDCHF', 'USDJPY')
```

Теперь перейдем к непосредственному извлечению свечных данных. Это напоминает получение тиковых данных, за тем лишь исключением, что необходимо указывать период, или толщину свечи (т1 соответствует одной минуте, H1 — одному часу, D1 — одному дню).

```
In [17]: start = dt.datetime(2018, 5, 1)
        stop = dt.datetime(2018, 6, 30)
```

```
In [18]: period = 'H1' ❶
```

```
In [19]: candles = cdr('EURUSD', start, stop, period)
```

```
In [20]: data = candles.get_data()
```

```
In [21]: data.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1080 entries, 2018-04-29 21:00:00 to
                2018-06-29 20:00:00
Data columns (total 8 columns):
BidOpen      1080 non-null float64
BidHigh      1080 non-null float64
BidLow       1080 non-null float64
BidClose     1080 non-null float64
AskOpen      1080 non-null float64
AskHigh      1080 non-null float64
AskLow       1080 non-null float64
AskClose     1080 non-null float64
dtypes: float64(8)
memory usage: 75.9 KB
```

```
In [22]: data[data.columns[:4]].tail() ❷
Out[22]:
```

	<b>BidOpen</b>	<b>BidHigh</b>	<b>BidLow</b>	<b>BidClose</b>
<b>2018-06-29 16:00:00</b>	1.16768	1.16820	1.16731	1.16769
<b>2018-06-29 17:00:00</b>	1.16769	1.16826	1.16709	1.16781
<b>2018-06-29 18:00:00</b>	1.16781	1.16816	1.16668	1.16684
<b>2018-06-29 19:00:00</b>	1.16684	1.16792	1.16638	1.16774
<b>2018-06-29 20:00:00</b>	1.16774	1.16904	1.16758	1.16816

```
In [23]: data[data.columns[4:]].tail() ❸
```

Out[23]:	AskOpen	AskHigh	AskLow	AskClose
2018-06-29 16:00:00	1.16769	1.16820	1.16732	1.16771
2018-06-29 17:00:00	1.16771	1.16827	1.16711	1.16782
2018-06-29 18:00:00	1.16782	1.16817	1.16669	1.16686
2018-06-29 19:00:00	1.16686	1.16794	1.16640	1.16775
2018-06-29 20:00:00	1.16775	1.16907	1.16760	1.16861

- ❶ Выбор периода (часовые интервалы).
- ❷ Цены спроса (*бид*): открытие, максимум, минимум, закрытие.
- ❸ Цены предложения (*аск*) открытие, максимум, минимум, закрытие.

Наконец, построим графики средних спредов закрытия и двух скользящих средних (рис. 14.2).

```
In [24]: data['MidClose'] = data[['BidClose', 'AskClose']].mean(axis=1) ❶
```

```
In [25]: data['SMA1'] = data['MidClose'].rolling(30).mean() ❷
         data['SMA2'] = data['MidClose'].rolling(100).mean() ❸
```

```
In [26]: data[['MidClose', 'SMA1', 'SMA2']].plot(figsize=(10, 6));
```

- ❶ Вычисление средних спредов закрытия.
- ❷ Вычисление двух скользящих средних (SMA): для более короткого и более длинного интервала.

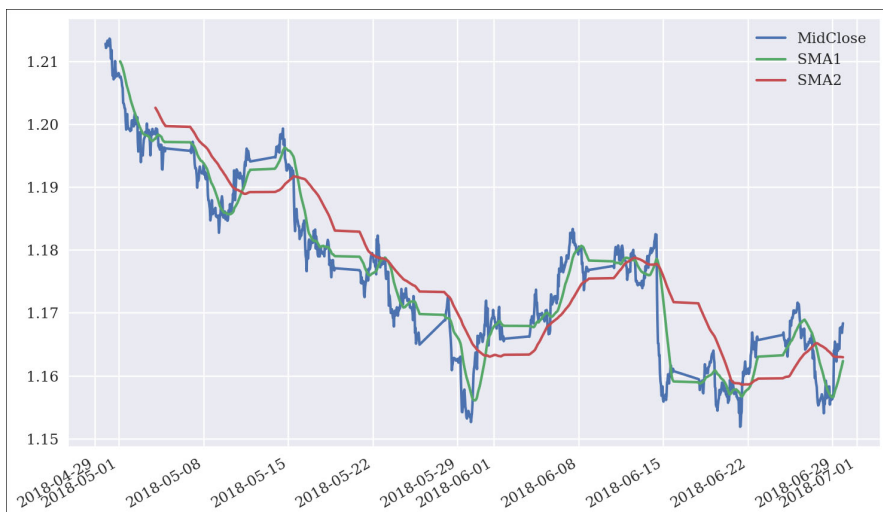


Рис. 14.2. Исторические почасовые средние спреды закрытия и два скользящих средних для валютной пары EUR/USD

## Работа с программным интерфейсом FXCM

Если в предыдущем разделе речь шла о непосредственной загрузке предварительно упакованных исторических тиковых и свечных данных с сервера FXCM, то в этом разделе будет показано, как загрузить исторические данные через программный интерфейс (API). Для этого нам понадобится объект подключения. Сначала мы импортируем пакет `fxcmpy`, затем создадим объект подключения (инициализируется уникальным API-токеном) и посмотрим доступные финансовые инструменты.

```
In [27]: import fxcmpy
```

```
In [28]: fxcmpy.__version__
```

```
Out[28]: '1.1.33'
```

```
In [29]: api = fxcmpy.fxcmpy(config_file='../fxcm.cfg') ❶
```

```
In [30]: instruments = api.get_instruments()
```

```
In [31]: print(instruments)
['EUR/USD', 'XAU/USD', 'GBP/USD', 'UK100', 'USDOLLAR',
 'XAG/USD', 'GER30', 'FRA40', 'USD/CNH', 'EUR/JPY',
 'USD/JPY', 'CHN50', 'GBP/JPY', 'AUD/JPY', 'CHF/JPY',
 'USD/CHF', 'GBP/CHF', 'AUD/USD', 'EUR/AUD', 'EUR/CHF',
 'EUR/CAD', 'EUR/GBP', 'AUD/CAD', 'NZD/USD', 'USD/CAD',
 'CAD/JPY', 'GBP/AUD', 'NZD/JPY', 'US30', 'GBP/CAD',
 'SOYF', 'GBP/NZD', 'AUD/NZD', 'USD/SEK', 'EUR/SEK',
 'EUR/NOK', 'USD/NOK', 'USD/MXN', 'AUD/CHF', 'EUR/NZD',
 'USD/ZAR', 'USD/HKD', 'ZAR/JPY', 'BTC/USD', 'USD/TRY',
 'EUR/TRY', 'NZD/CHF', 'CAD/CHF', 'NZD/CAD', 'TRY/JPY',
 'AUS200', 'ESP35', 'HKG33', 'JPN225', 'NAS100', 'SPX500',
 'Copper', 'EUSTX50', 'USOil', 'UKOil', 'NGAS', 'Bund']
```

- ❶ Подключение к программному интерфейсу (укажите правильный путь/имя файла).

## Получение исторических данных

После подключения к программному интерфейсу можно приступать к получению данных за необходимые интервалы времени. Это реализуется с помощью одного-единственного вызова метода `get_candels()`, параметр `period` которого может быть равен `m1`, `m5`, `m15`, `m30`, `H1`, `H2`, `H3`, `H4`, `H6`, `H8`, `D1`, `W1` или `M1`.

Ниже приведено несколько примеров. На рис. 14.3 показан график поминутных цен закрытия предложений для инструмента (валютной пары) EUR/USD.

```
In [32]: candles = api.get_candles('USD/JPY', period='D1', number=10) ❶
```

```
In [33]: candles[candles.columns[:4]] ❶
```

```
Out[33]:
```

	date	bidopen	bidclose	bidhigh	bidlow
	2018-10-08 21:00:00	113.760	113.219	113.937	112.816
	2018-10-09 21:00:00	113.219	112.946	113.386	112.863
	2018-10-10 21:00:00	112.946	112.267	113.281	112.239
	2018-10-11 21:00:00	112.267	112.155	112.528	111.825
	2018-10-12 21:00:00	112.155	112.200	112.491	111.873
	2018-10-14 21:00:00	112.163	112.130	112.270	112.109
	2018-10-15 21:00:00	112.130	111.758	112.230	111.619
	2018-10-16 21:00:00	112.151	112.238	112.333	111.727
	2018-10-17 21:00:00	112.238	112.636	112.670	112.009
	2018-10-18 21:00:00	112.636	112.168	112.725	111.942

```
In [34]: candles[candles.columns[4:]] ❶
```

```
Out[34]:
```

	date	askopen	askclose	askhigh	asklow	tickqty
	2018-10-08 21:00:00	113.840	113.244	113.950	112.827	184835
	2018-10-09 21:00:00	113.244	112.970	113.399	112.875	321755
	2018-10-10 21:00:00	112.970	112.287	113.294	112.265	329174
	2018-10-11 21:00:00	112.287	112.175	112.541	111.835	568231
	2018-10-12 21:00:00	112.175	112.243	112.504	111.885	363233
	2018-10-14 21:00:00	112.219	112.181	112.294	112.145	581
	2018-10-15 21:00:00	112.181	111.781	112.243	111.631	322304
	2018-10-16 21:00:00	112.163	112.271	112.345	111.740	253420
	2018-10-17 21:00:00	112.271	112.664	112.682	112.022	542166
	2018-10-18 21:00:00	112.664	112.237	112.738	111.955	369012

```
In [35]: start = dt.datetime(2017, 1, 1) ❷
```

```
end = dt.datetime(2018, 1, 1) ❷
```

```
In [36]: candles = api.get_candles('EUR/GBP', period='D1',  
start=start, stop=end) ❷
```

```
In [37]: candles.info() ❷
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
DatetimeIndex: 309 entries, 2017-01-03 22:00:00 to
```

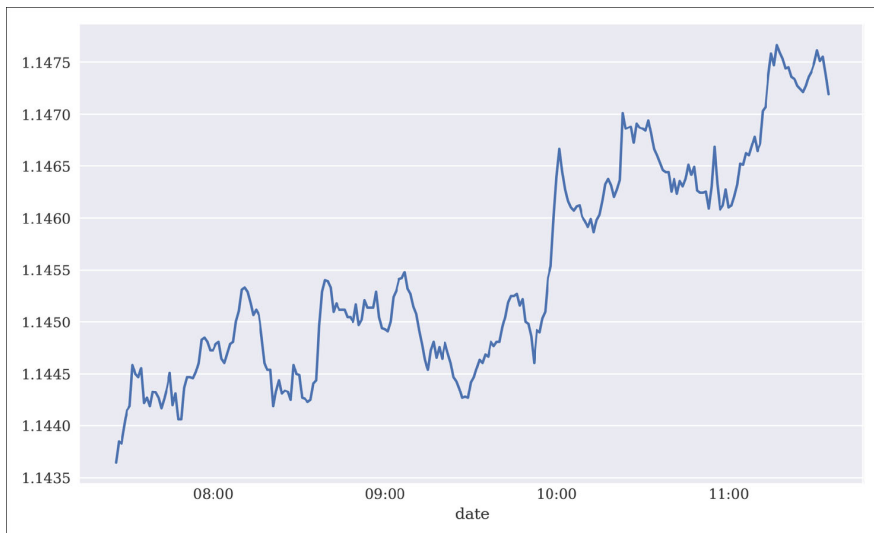
```
2018-01-01 22:00:00
```

```
Data columns (total 9 columns):
bidopen    309 non-null float64
bidclose   309 non-null float64
bidhigh    309 non-null float64
bidlow     309 non-null float64
askopen    309 non-null float64
askclose   309 non-null float64
askhigh    309 non-null float64
asklow     309 non-null float64
tickqty    309 non-null int64
dtypes: float64(8), int64(1)
memory usage: 24.1 KB
```

```
In [38]: candles = api.get_candles('EUR/USD', period='m1',
                                   number=250) ❸
```

```
In [39]: candles['askclose'].plot(figsize=(10, 6))
```

- ❶ Получение суточных свечей за 10 последних дней.
- ❷ Получение суточных свечей за весь год.
- ❸ Получение самых последних минутных свечей.



**Рис. 14.3.** Исторические цены закрытия предложений для валютной пары EUR/USD (минутные свечи)

## Получение потоковых данных

Исторические данные важны для тестирования стратегий алгоритмической торговли, но для развертывания автоматизированных торговых систем необходим непрерывный доступ к данным *реального времени* (потоковым данным). Программный интерфейс FXCM позволяет подписаться на потоковые данные для всех финансовых инструментов. Оболочечный пакет `fxcmру` поддерживает работу с потоковыми данными через так называемые *функции обратного вызова*, предоставляемые пользователем.

В следующем примере создается простая функция обратного вызова, которая отображает выбранные элементы извлекаемого набора данных. Эта функция применяется к потоковым данным финансового инструмента, на который оформлена подписка (в нашем случае EUR/USD).

```
In [40]: def output(data, dataframe):  
        print('%3d | %s | %s | %6.5f, %6.5f' %  
              (len(dataframe), data['Symbol'],  
               pd.to_datetime(int(data['Updated'])), unit='ms'),  
              data['Rates'][0], data['Rates'][1])) ❶  
  
In [41]: api.subscribe_market_data('EUR/USD', (output,)) ❷  
1 | EUR/USD | 2018-10-19 11:36:39.735000 | 1.14694, 1.14705  
2 | EUR/USD | 2018-10-19 11:36:39.776000 | 1.14694, 1.14706  
3 | EUR/USD | 2018-10-19 11:36:40.714000 | 1.14695, 1.14707  
4 | EUR/USD | 2018-10-19 11:36:41.646000 | 1.14696, 1.14708  
5 | EUR/USD | 2018-10-19 11:36:41.992000 | 1.14696, 1.14709  
6 | EUR/USD | 2018-10-19 11:36:45.131000 | 1.14696, 1.14708  
7 | EUR/USD | 2018-10-19 11:36:45.247000 | 1.14696, 1.14709  
  
In [42]: api.get_last_price('EUR/USD') ❸  
Out[42]: Bid    1.14696  
         Ask    1.14709  
         High   1.14775  
         Low    1.14323  
         Name: 2018-10-19 11:36:45.247000, dtype: float64
```

```
In [43]: api.unsubscribe_market_data('EUR/USD') ❹  
8 | EUR/USD | 2018-10-19 11:36:48.239000 | 1.14696, 1.14708
```

- ❶ Функция обратного вызова, которая выводит отдельные элементы извлекаемого набора данных.

- ❷ Подписка на конкретный поток данных реального времени. Данные обрабатываются в асинхронном режиме, пока пользователь не отпишется от них.
- ❸ Пока действует подписка, метод `get_last_price()` возвращает последний доступный набор данных.
- ❹ Отмена подписки на потоковые данные.



### Функции обратного вызова

Функции обратного вызова — гибкое средство обработки потоковых данных, поступающих в режиме реального времени. Их можно использовать как для выполнения простых операций (например, вывод поступающих данных), так и для решения более сложных задач, включая генерацию торговых сигналов, основанных на алгоритмах интернет-трейдинга (глава 16).

## Размещение заявок

Программный интерфейс FXCM позволяет управлять любыми типами заявок, доступными также через торговое приложение FXCM (включая лимитные заявки и стоп-лоссы)<sup>2</sup>. Однако в следующем примере демонстрируется лишь, как создавать простые рыночные заявки, так как этого достаточно для знакомства с принципами алгоритмической торговли. Сначала мы убеждаемся в отсутствии открытых позиций, а затем открываем несколько разных позиций с помощью метода `create_market_buy_order()`.

```
In [44]: api.get_open_positions() ❶
```

```
Out[44]: Empty DataFrame
         Columns: []
         Index: []
```

```
In [45]: order = api.create_market_buy_order('EUR/USD', 10) ❷
```

```
In [46]: sel = ['tradeId', 'amountK', 'currency',
               'grossPL', 'isBuy'] ❸
```

```
In [47]: api.get_open_positions()[sel] ❹
```

```
Out[47]:
```

	tradeId	amountK	currency	grossPL	isBuy
0	132607899	10	EUR/USD	0.17436	True

<sup>2</sup> Детали можно узнать в документации (<http://fxcm.py.tpq.io/>).



```
In [48]: order = api.create_market_buy_order('EUR/GBP', 5) ❹
```

```
In [49]: api.get_open_positions()[sel]
```

```
Out[49]:
```

	tradeId	amountK	currency	grossPL	isBuy
0	132607899	10	EUR/USD	0.17436	True
1	132607928	5	EUR/GBP	-1.53367	True

- ❶ Отображение сведений об открытых позициях текущей учетной записи.
- ❷ Открытие позиции размером 10 000 для валютной пары EUR/USD<sup>3</sup>.
- ❸ Отображение открытых позиций только для выбранных элементов.
- ❹ Открытие еще одной позиции размером 5 000 для валютной пары EUR/GBP.

Функция `create_market_buy_order()` создает заявку на покупку, а функция `create_market_sell_order()` — заявку на продажу. Существуют также более общие методы, позволяющие закрывать позиции, как показано ниже.

```
In [50]: order = api.create_market_sell_order('EUR/USD', 3) ❶
```

```
In [51]: order = api.create_market_buy_order('EUR/GBP', 5) ❷
```

```
In [52]: api.get_open_positions()[sel] ❸
```

```
Out[52]:
```

	tradeId	amountK	currency	grossPL	isBuy
0	132607899	10	EUR/USD	0.17436	True
1	132607928	5	EUR/GBP	-1.53367	True
2	132607930	3	EUR/USD	-1.33369	False
3	132607932	5	EUR/GBP	-1.64728	True

```
In [53]: api.close_all_for_symbol('EUR/GBP') ❹
```

```
In [54]: api.get_open_positions()[sel]
```

```
Out[54]:
```

	tradeId	amountK	currency	grossPL	isBuy
0	132607899	10	EUR/USD	0.17436	True
1	132607930	3	EUR/USD	-1.33369	False

```
In [55]: api.close_all() ❺
```

---

<sup>3</sup> Размер позиции указывается в тысячах единиц. Также учтите, что для разных учетных записей могут устанавливаться разные *маржинальные требования* (<https://www.fxcm.com/uk/accounts/forex-cfd-leverage/>). Это означает, что для одной и той же позиции может потребоваться больший или меньший залоговый депозит, в зависимости от плеча финансового рычага. В случае необходимости уменьшите соответствующие значения при создании заявок.

```
In [56]: api.get_open_positions()
Out[56]: Empty DataFrame
         Columns: []
         Index: []
```

- ❶ Понижение позиции в валютной паре EUR/USD.
- ❷ Повышение позиции в валютной паре EUR/GBP.
- ❸ Теперь инструмент EUR/GBP представлен двумя открытыми позициями покупки, тогда как для инструмента EUR/USD имеется позиция покупки и позиция продажи.
- ❹ Метод `close_all_for_symbol()` закрывает все позиции для указанного символа.
- ❺ Метод `close_all()` закрывает все открытые позиции.

## Учетные данные

Помимо информации об открытых позициях программный интерфейс FXCM позволяет получать общие сведения об учетной записи. Можно, например, просмотреть идентификатор учетной записи, заданной по умолчанию (если есть несколько учетных записей), а также узнать величину капитала и маржи.

```
In [57]: api.get_default_account() ❶
Out[57]: 1090495
```

```
In [58]: api.get_accounts().T ❷
Out[58]:
```

	0
accountId	1090495
accountName	01090495
balance	4915.2
dayPL	-41.97
equity	4915.2
grossPL	0
hedging	Y
mc	N
mcDate	
ratePrecision	0
t	6
usableMargin	4915.2
usableMargin3	4915.2
usableMargin3Perc	100

<code>usableMarginPerc</code>	100
<code>usdMr</code>	0
<code>usdMr</code>	3

- ❶ Вывод идентификатора учетной записи, заданной по умолчанию.
- ❷ Отображение финансовой ситуации и отдельных показателей для всех учетных записей.

## Резюме

В этой главе, посвященной REST-совместимому программному интерфейсу алгоритмической торговли FXCM, рассматривались следующие темы:

- конфигурирование программного интерфейса;
- получение исторических тиковых данных;
- получение исторических свечных данных;
- получение потоковых данных в режиме реального времени;
- размещение рыночных заявок на покупку и продажу;
- просмотр информации об учетной записи.

Разумеется, функциональные возможности программного интерфейса FXCM и оболочечного пакета `fxcmru` значительно шире, но приведенных в этой главе сведений вполне достаточно для того, чтобы начать заниматься алгоритмической торговлей.

## Дополнительные ресурсы

Дополнительные сведения о программном интерфейсе FXCM и оболочечном пакете `fxcmru` содержатся в документации:

- программный интерфейс FXCM (<https://fxcm.github.io/rest-api-docs/>);
- пакет `fxcmru` (<https://fxcmru.tpq.io/>).

Специализированный онлайн-курс, посвященный использованию Python для алгоритмической торговли, доступен на сайте <http://certificate.tpq.io>.

---

## Торговые стратегии

Они были достаточно глупы, чтобы поверить, будто вы сможете предсказать будущее, заглянув в прошлое.

*The Economist*

Эта глава посвящена векторизованному тестированию алгоритмических торговых стратегий на исторических данных. Термин *алгоритмическая торговая стратегия* используется для описания любого типа торговой стратегии, которая основана на алгоритме, способном самостоятельно создавать открытые длинные, открытые короткие и закрытые позиции по финансовым инструментам без участия трейдера. Данному определению соответствует, например, такой простой алгоритм, как “каждые пять минут переходить от длинной к закрытой позиции по акциям компании Apple”. Для целей главы мы будем представлять алгоритмическую торговую стратегию в виде кода Python, который при наличии новых данных принимает решение о покупке или продаже финансового инструмента, создавая длинные, короткие или закрытые позиции.

В главе не дается обзор всех доступных стратегий алгоритмической торговли (об этом можно прочитать в специализированной литературе, упомянутой в разделе “Дополнительные ресурсы”). Вместо этого мы сконцентрируемся на технических аспектах *векторизованного тестирования* нескольких таких стратегий на исторических данных. В векторизованном подходе финансовые данные, на которых проверяется стратегия, рассматриваются как единое целое, и соответствующие операции выполняются сразу над всем объектом `ndarray` библиотеки NumPy или объектом `DataFrame` библиотеки pandas<sup>1</sup>.

Еще одна тема главы — применение алгоритмов машинного и глубокого обучения для реализации алгоритмических торговых стратегий. Мы будем обучать алгоритмы классификации на исторических данных, чтобы иметь

---

<sup>1</sup> Альтернативный подход — *событийное тестирование* торговых стратегий, в рамках которого появление новых рыночных данных моделируется путем явного прохода по каждой новой точке данных.

возможность предсказывать будущие колебания рынка. Обычно для этого требуется преобразовать финансовые данные из вещественных значений в относительно небольшое количество категориальных значений<sup>2</sup>, что позволяет задействовать методы распознавания образов.

В главе рассматриваются следующие темы.

#### *Простое скользящее среднее*

В этом разделе рассматриваются алгоритмические торговые стратегии, основанные на простом скользящем среднем, и способы их тестирования на исторических данных.

#### *Гипотеза случайного блуждания*

В этом разделе мы рассмотрим гипотезу случайного блуждания.

#### *Линейная регрессия по методу наименьших квадратов*

Этот раздел посвящен применению регрессии по методу наименьших квадратов для разработки алгоритмической торговой стратегии.

#### *Кластеризация*

В этом разделе мы рассмотрим применение технологий машинного обучения без учителя для разработки алгоритмической торговой стратегии.

#### *Частотный подход*

В этом разделе рассматривается простой частотный подход к алгоритмической торговле.

#### *Классификация*

В этом разделе мы рассмотрим применение алгоритмов классификации в алгоритмической торговле.

#### *Глубокие нейронные сети*

В этом разделе мы поговорим о применении глубоких нейронных сетей в алгоритмической торговле.

## Простое скользящее среднее

Торговые стратегии на основе простого скользящего среднего (Simple Moving Average — SMA) распространены уже достаточно давно (см., например, работу Брока и соавторов [2]). Несмотря на то что трейдеры применяют

---

<sup>2</sup> При работе с вещественными значениями каждый шаблон может оказаться уникальным или, по крайней мере, довольно редким, что затрудняет обучение алгоритма и не позволяет делать никаких выводов из факта распознавания такого образа.

SMA преимущественно в ручной торговле, этот метод может также стать основой для формирования простой алгоритмической стратегии. В данном разделе мы будем использовать SMA для векторизованного тестирования алгоритмических стратегий на исторических данных. Мы продолжим пример технического анализа, начатый в главе 8.

## Импорт данных

Начнем с импорта модулей.

```
In [1]: import numpy as np
import pandas as pd
import datetime as dt
from pylab import mpl, plt
```

```
In [2]: plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

Далее необходимо загрузить необработанные данные и выбрать финансовый временной ряд для единственного тикера, представляющего акции компании Apple, Inc. (AAPL.O). В этом разделе мы будем анализировать данные на конец торгов. Внутриденные данные будут обрабатываться в последующих разделах.

```
In [3]: raw = pd.read_csv('../source/tr_eikon_eod_data.csv',
index_col=0, parse_dates=True)
```

```
In [4]: raw.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
Data columns (total 12 columns):
AAPL.O      2138 non-null float64
MSFT.O      2138 non-null float64
INTC.O      2138 non-null float64
AMZN.O      2138 non-null float64
GS.N        2138 non-null float64
SPY         2138 non-null float64
.SPX        2138 non-null float64
.VIX        2138 non-null float64
EUR=        2216 non-null float64
XAU=        2211 non-null float64
GDV         2138 non-null float64
GLD         2138 non-null float64
```

```
dtypes: float64(12)
memory usage: 225.1 KB
```

```
In [5]: symbol = 'AAPL.O'
```

```
In [6]: data = (
    pd.DataFrame(raw[symbol])
    .dropna()
)
```

## Торговая стратегия

Рассчитаем величину скользящего среднего для двух разных размеров окна. На рис. 15.1 представлены три временных ряда.

```
In [7]: SMA1 = 42 ❶
    SMA2 = 252 ❷
```

```
In [8]: data['SMA1'] = data[symbol].rolling(SMA1).mean() ❶
    data['SMA2'] = data[symbol].rolling(SMA2).mean() ❷
```

```
In [9]: data.plot(figsize=(10, 6));
```

- ❶ Вычисление значений *краткосрочного* SMA.
- ❷ Вычисление значений *долгосрочного* SMA.



Рис. 15.1. Цена акций компании Apple и два простых скользящих средних

Наконец, приступим к формированию торговой стратегии. Правила трейдинга будут такими:

- если линия краткосрочного SMA оказывается выше линии долгосрочного SMA, открываем *длинную* позицию ( $= +1$ );
- если линия краткосрочного SMA оказывается ниже линии долгосрочного SMA, открываем *короткую* позицию ( $= -1$ ).

Соответствующие графики показаны на рис. 15.2.



Рис. 15.2. Цена акций компании Apple, два скользящих средних и получаемые позиции

```
In [10]: data.dropna(inplace=True)
```

```
In [11]: data['Позиция'] = np.where(data['SMA1'] > data['SMA2'],
                                     1, -1) ❶
```

```
In [12]: data.tail()
```

```
Out[12]:
```

	AAPL.O	SMA1	SMA2	Позиция
2018-06-25	182.17	185.606190	168.265556	1
2018-06-26	184.43	186.087381	168.418770	1
2018-06-27	184.16	186.607381	168.579206	1
2018-06-28	185.50	187.089286	168.736627	1
2018-06-29	185.11	187.470476	168.901032	1



```
In [13]: ax = data.plot(secondary_y='Позиция', figsize=(10, 6),
                        mark_right=False)
ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

- ❶ Функция `np.where(cond, a, b)` поэлементно проверяет условие `cond`, возвращая `a`, если условие истинно (`True`), и `b` — в противном случае.

Как видите, результаты совпадают с теми, которые мы получили в главе 8. Единственное, что мы не рассматривали, — это вопрос о том, дает ли соблюдение алгоритмической торговой стратегии какие-то преимущества по сравнению с эталонным случаем, когда на протяжении всего периода мы удерживаем длинную позицию по акциям Apple. Поскольку в рамках стратегии мы лишь дважды переходим к коротким позициям, только на этих двух отрезках могут проявиться различия в доходности.

## Векторизованное тестирование на исторических данных

Теперь можно перейти к векторизованному тестированию торговой стратегии на исторических данных. Сначала необходимо вычислить логарифмические доходности, которые затем умножаются на весовые значения позиций (+1 или -1). Такой простой подход возможен, поскольку длинная позиция по ценным бумагам Apple обеспечивает положительную, а короткая позиция — отрицательную доходность. Для определения общей доходности остается только суммировать значения логарифмической доходности акций Apple и алгоритмической торговой стратегии, основанной на SMA, и применить к полученному результату экспоненциальную функцию.

```
In [14]: data['Доходность'] = np.log(data[symbol] /
                                     data[symbol].shift(1)) ❶
```

```
In [15]: data['Стратегия'] = data['Позиция'].shift(1) *
                                     data['Доходность'] ❷
```

```
In [16]: data.round(4).head()
```

```
Out[16]:
```

	AAPL.O	SMA1	SMA2	Позиция	\
Date					
2010-12-31	46.0800	45.2810	37.1207	1	
2011-01-03	47.0814	45.3497	37.1862	1	
2011-01-04	47.3271	45.4126	37.2525	1	
2011-01-05	47.7142	45.4661	37.3223	1	
2011-01-06	47.6757	45.5226	37.3921	1	

	Доходность	Стратегия
Date		
2010-12-31	NaN	NaN
2011-01-03	0.0215	0.0215
2011-01-04	0.0052	0.0052
2011-01-05	0.0081	0.0081
2011-01-06	-0.0008	-0.0008

```
In [17]: data.dropna(inplace=True)
```

```
In [18]: np.exp(data[['Доходность', 'Стратегия']].sum()) ❸
```

```
Out[18]: Доходность    4.017148
          Стратегия    5.811299
          dtype: float64
```

```
In [19]: data[['Доходность', 'Стратегия']].std() * 252 ** 0.5 ❹
```

```
Out[19]: Доходность    0.250571
          Стратегия    0.250407
          dtype: float64
```

- ❶ Вычисление логарифмической доходности акций Apple (эталонная инвестиция).
- ❷ Умножение весового коэффициента позиции, заданного с суточным смещением, на значение логарифмической доходности; временной сдвиг позволяет избежать смещенного прогноза<sup>3</sup>.
- ❸ Суммирование логарифмических доходностей эталонной инвестиции и алгоритмической торговой стратегии с последующим применением экспоненциальной функции для вычисления абсолютной доходности.
- ❹ Вычисление годовой волатильности для эталонной инвестиции и алгоритмической торговой стратегии.

Цифры говорят о том, что выбранная алгоритмическая торговая стратегия оказывается эффективнее эталонной инвестиции, предполагающей пассивное удержание акций Apple. Тип и характер торговой стратегии обеспечивают такую же годовую волатильность, как и у эталонной инвестиции, благодаря чему стратегия показывает лучшие результаты с поправкой на риск.

---

<sup>3</sup> Идея состоит в том, что алгоритм может открыть позицию по акциям Apple только на основании *сегодняшних рыночных данных* (до завершения торгов). Соответственно, доходность позиции рассчитывается для *следующего дня*.

Чтобы нагляднее увидеть разницу в доходности, рассмотрим рис. 15.3, на котором сравниваются графики доходности акций компании Apple и алгоритмической торговой стратегии.

```
In [20]: ax = data[['Доходность', 'Стратегия']].cumsum(  
          ).apply(np.exp).plot(figsize=(10, 6))  
        data['Позиция'].plot(ax=ax, secondary_y='Позиция',  
                              style='--')  
        ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```



*Рис. 15.3. Изменение доходности акций Apple и торговой стратегии на основе SMA*



### Упрощения

Векторизованное тестирование торговой стратегии на исторических данных предполагает ряд упрощений. Прежде всего, здесь не учитываются транзакционные издержки (фиксированные комиссии, кредитные расходы и т.п.). Это приемлемо только в торговых стратегиях, предполагающих совершение всего нескольких сделок за несколько лет. Также предполагается, что все сделки совершаются по ценам закрытия, зафиксированным на конец торгов. В более реалистичном подходе все эти и многие другие микро-структурные рыночные факторы нужно обязательно учитывать.

## Оптимизация

Возникает логичный вопрос о том, насколько оправдан выбор параметров SMA1=42 и SMA2=252. Понятно, что при прочих равных условиях инвесторы предпочитают более высокую доходность более низкой. Поэтому можно попытаться подобрать такие параметры, которые обеспечат максимальную доходность за требуемый период времени. Одно из решений — метод грубой силы, при котором вся процедура векторизованного тестирования повторяется для всех возможных комбинаций параметров, после чего выбирается лучший из результатов. Ниже реализован именно такой подход.

```
In [21]: from itertools import product
```

```
In [22]: sma1 = range(20, 61, 4) ❶  
         sma2 = range(180, 281, 10) ❷
```

```
In [23]: results = pd.DataFrame()  
         for SMA1, SMA2 in product(sma1, sma2): ❸  
             data = pd.DataFrame(raw[symbol])  
             data.dropna(inplace=True)  
             data['Доходность'] = np.log(data[symbol] /  
                                         data[symbol].shift(1))  
             data['SMA1'] = data[symbol].rolling(SMA1).mean()  
             data['SMA2'] = data[symbol].rolling(SMA2).mean()  
             data.dropna(inplace=True)  
             data['Позиция'] = np.where(data['SMA1'] >  
                                         data['SMA2'], 1, -1)  
             data['Стратегия'] = data['Позиция'].shift(1) *  
                                 data['Доходность']  
             data.dropna(inplace=True)  
             perf = np.exp(data[['Доходность', 'Стратегия']].sum())  
             results = results.append(pd.DataFrame(  
                 {'SMA1': SMA1, 'SMA2': SMA2,  
                  'MARKET': perf['Доходность'],  
                  'STRATEGY': perf['Стратегия'],  
                  'OUT': perf['Стратегия'] - perf['Доходность']},  
                 index=[0]), ignore_index=True) ❹
```

- ❶ Диапазон значений для параметра SMA1.
- ❷ Диапазон значений для параметра SMA2.
- ❸ Перебор всех комбинаций параметров SMA1 и SMA2.
- ❹ Сохранение результатов векторизованного тестирования в объекте DataFrame.

В следующем коде выводятся семь наиболее эффективных комбинаций параметров. Ранжирование проводится по разнице между доходностью алгоритмической торговой стратегии и эталонной инвестиции. Учтите, что доходность эталонной инвестиции тоже варьируется, поскольку значение параметра SMA2 влияет на длительность временного интервала и объем данных, для которых выполняется векторизованное тестирование.

```
In [24]: results.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 121 entries, 0 to 120
Data columns (total 5 columns):
SMA1      121 non-null int64
SMA2      121 non-null int64
MARKET    121 non-null float64
STRATEGY  121 non-null float64
OUT       121 non-null float64
dtypes: float64(3), int64(2)
memory usage: 4.8 KB

In [25]: results.sort_values('OUT', ascending=False).head(7)
Out[25]:
```

	SMA1	SMA2	MARKET	STRATEGY	OUT
56	40	190	4.650342	7.175173	2.524831
39	32	240	4.045619	6.558690	2.513071
59	40	220	4.220272	6.544266	2.323994
46	36	200	4.074753	6.389627	2.314874
55	40	180	4.574979	6.857989	2.283010
70	44	220	4.220272	6.469843	2.249571
101	56	200	4.074753	6.319524	2.244772

Согласно полученным результатам оптимальными будут параметры SMA1 = 40 и SMA2 = 190, обеспечивающие 250-процентную разницу в доходности. Однако оптимизация сильно зависит от используемого набора данных, что может привести к переобучению модели. Более строгий подход предполагает выполнять оптимизацию параметров на одном наборе данных (обучающем, в пределах выборки), а векторизованное тестирование — на другом наборе (тестовом, вне выборки).



### Переобучение

Как правило, любой метод оптимизации, подгонки или обучения модели в контексте построения алгоритмических торговых стратегий подвержен проблеме *переобучения*. Это означает, что существует риск выбрать такие параметры, которые будут давать

(исключительно) хорошие результаты для исходного набора данных и (исключительно) плохие результаты для других наборов данных, встречающихся на практике.

## Гипотеза случайного блуждания

В предыдущем разделе вы узнали, как выполнить векторизованное тестирование алгоритмических торговых стратегий на исторических данных (*ретроспективное тестирование*). Тестирование отдельной стратегии на единственном временном ряде, представляющем исторические данные о ценах акций Apple на конец торгов, показало, что ее доходность выше доходности эталонной инвестиции, предполагающей удерживание длинных позиций по акциям Apple в течение аналогичного периода времени.

Но это, скорее, частный случай. Результаты тестирования противоречат *гипотезе случайного блуждания* (Random Walk Hypothesis — RWH), которая утверждает, что такого рода прогнозы не должны предсказывать никакого прироста доходности. Согласно гипотезе цены на финансовых рынках подвержены случайному блужданию и в непрерывном времени описываются моделью арифметического броуновского движения без смещения (дрейфа). Ожидаемое значение такого броуновского движения в любой будущей временной точке равно его значению сегодня<sup>4</sup>. Как следствие, в гипотезе случайного блуждания наилучшим предсказанием будущей цены, рассчитываемой по методу наименьших квадратов, будет сегодняшняя цена.

Такой вывод подкрепляется следующей цитатой.

В течение многих лет экономисты, математики и специалисты по финансам занимались разработкой и тестированием моделей поведения биржевых котировок. Результатом их исследований стало появление теории случайных блужданий. Эта теория подвергает сомнению многие другие методы, применяемые для описания и предсказания поведения биржевых котировок и получившие широкое распространение вне академической среды. Как будет показано далее, если теория случайных блужданий достаточно точно описывает действительные биржевые процессы, то различные “околонаучные” методики предсказания котировок акций оказываются совершенно бесполезными.

Юджин Фама [3]

Гипотеза случайного блуждания прекрасно согласуется с *гипотезой эффективного рынка* (Efficient Markets Hypothesis — EMH), которая в целом утверждает

---

<sup>4</sup> Формальное определение и подробное описание процессов случайного блуждания и броуновского движения приведены в книге Бакстера и Ренни [1].

ет, что рыночные котировки отражают “всю доступную информацию”. Различают три формы эффективности рынка — *слабую, среднюю и сильную*, — каждая из которых очерчивает свои рамки “всей доступной информации”. С формальной точки зрения такое определение основывается на концепции информационного множества в теории игр, что подтверждается следующей цитатой.

Рынок эффективен в отношении информационного множества  $S$ , если невозможно получить прибыль, занимаясь торговлей на основе этого множества.

Майкл Дженсен [4]

В Python гипотезу случайного блуждания можно проверить для конкретного случая. Берется финансовый временной ряд, содержащий исторические данные о рыночных ценах, и для него создается несколько *смещенных* версий, допустим, пять. Далее к ним применяется регрессия по методу наименьших квадратов для прогнозирования рыночных цен. Основная идея заключается в том, что на основе вчерашней цены и цен за предыдущие четыре дня можно предсказать сегодняшнюю цену.

Ниже показано, как реализовать такой алгоритм на Python. В этом примере создается пять смещенных версий временного ряда, содержащего исторические данные об уровнях индекса S&P 500 на конец торгов.

```
In [26]: symbol = '.SPX'
```

```
In [27]: data = pd.DataFrame(raw[symbol])
```

```
In [28]: lags = 5
         cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)❶
             data[col] = data[symbol].shift(lag)❷
             cols.append(col)❸
```

```
In [29]: data.head(7)
```

```
Out[29]:
```

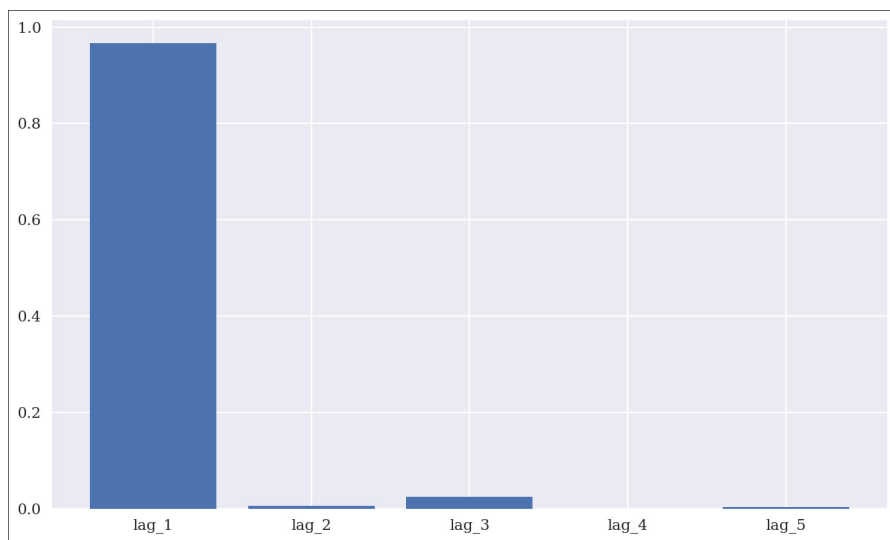
	.SPX	lag_1	lag_2	lag_3	lag_4	\
Date						
2010-01-01	NaN	NaN	NaN	NaN	NaN	
2010-01-04	1132.99	NaN	NaN	NaN	NaN	
2010-01-05	1136.52	1132.99	NaN	NaN	NaN	
2010-01-06	1137.14	1136.52	1132.99	NaN	NaN	
2010-01-07	1141.69	1137.14	1136.52	1132.99	NaN	
2010-01-08	1144.98	1141.69	1137.14	1136.52	1132.99	
2010-01-11	1146.98	1144.98	1141.69	1137.14	1136.52	

Date	lag_5
2010-01-01	NaN
2010-01-04	NaN
2010-01-05	NaN
2010-01-06	NaN
2010-01-07	NaN
2010-01-08	NaN
2010-01-11	1132.99

In [30]: `data.dropna(inplace=True)`

- ❶ Определение названия столбца для текущей копии набора данных.
- ❷ Создание смещенной версии набора данных о ценах.
- ❸ Создание массива имен столбцов для дальнейшего использования.

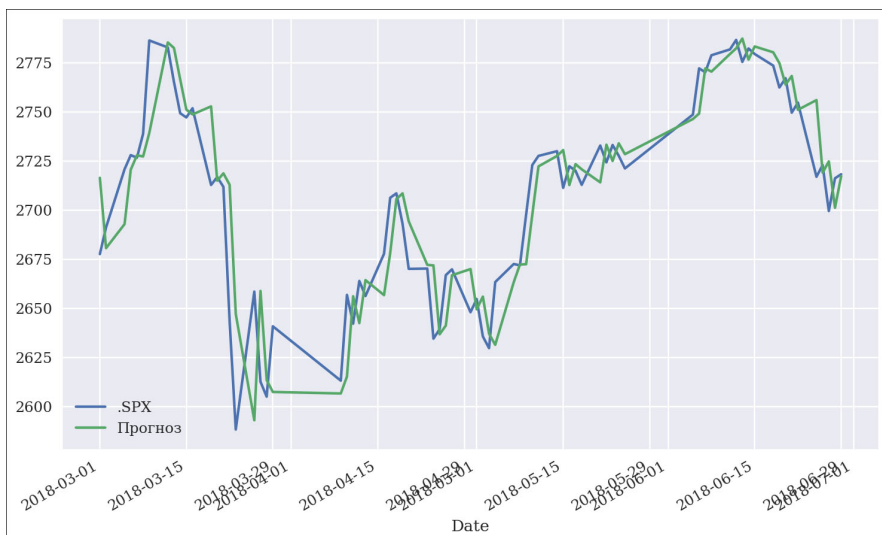
Регрессию по методу наименьших квадратов проще всего выполнить средствами библиотеки NumPy. Как показывает анализ регрессионных параметров, наибольшее влияние на точность предсказания рыночных цен оказывает ряд `lag_1`. Его вклад близок к 1. Вклады остальных четырех рядов близки к 0 (рис. 15.4).



*Рис. 15.4. Оптимальные регрессионные параметры при прогнозировании цен по методу наименьших квадратов*



При сравнении прогнозируемых значений, соответствующих оптимальным параметрам регрессии, с действительными значениями индекса S&P 500 становится очевидным, что прогноз строится преимущественно на основе ряда  $\text{lag}_1$ . На рис. 15.5 график предсказаний повторяет график исходного временного ряда, смещенного на один день вправо (с незначительными корректировками).



*Рис. 15.5. Уровни индекса S&P 500 и его прогнозируемые значения, полученные с помощью регрессии по методу наименьших квадратов*

В целом проведенный в этом разделе краткий анализ частично подтверждает справедливость гипотезы случайного блуждания и гипотезы эффективного рынка. Конечно, анализ был выполнен только для одного индекса, причем с довольно специфической параметризацией, но предложенную модель можно легко расширить, включив в нее несколько финансовых инструментов по нескольким классам активов, а также используя для них разное количество смещенных наборов данных. По сути, результаты будут теми же. В конце концов, обе гипотезы представляют собой финансовые теории, подкрепляемые большим количеством эмпирических данных. В этом смысле для доказательства любой алгоритмической торговой стратегии нужно сначала показать, что гипотеза случайного блуждания не применима в данном контексте, а такую задачу решить очень непросто.

## Линейная регрессия по методу наименьших квадратов

В этом разделе мы применим *линейную регрессию по методу наименьших квадратов*, чтобы спрогнозировать направление движения рынка на основе исторических данных о логарифмической доходности. Для простоты мы будем рассматривать всего два показателя. Первый из них (`lag_1`) представляет логарифмические доходности финансового временного ряда, смещенного на *один* день. Второй показатель (`lag_2`) описывает логарифмические доходности этого же временного ряда, смещенного на *два* дня. В отличие от цен логарифмические доходности в целом *неизменны*, что зачастую является необходимым условием для применения алгоритмов статистического анализа и машинного обучения.

Идея использования смещенных логарифмических доходностей заключается в том, что на их основе можно прогнозировать будущие доходности. Например, можно выдвинуть гипотезу, что после двух подряд движений вниз более вероятным оказывается движение вверх (возврат к среднему) или, наоборот, движение вниз (моментум, или тренд). Применение регрессионных методов позволяет формализовать подобные рассуждения.

### Данные

Вначале необходимо импортировать и подготовить набор данных. На рис. 15.6 показано частотное распределение исторических значений суточной логарифмической доходности валютной пары EUR/USD. На их основе будут получены необходимые финансовые показатели, а также метки.

```
In [3]: raw = pd.read_csv('../source/tr_eikon_eod_data.csv',
                        index_col=0, parse_dates=True).dropna()

In [4]: raw.columns
Out[4]: Index(['AAPL.0', 'MSFT.0', 'INTC.0', 'AMZN.0', 'GS.N',
              'SPY', '.SPX', '.VIX', 'EUR=', 'XAU=', 'GDX', 'GLD'],
             dtype='object')
```

```
In [5]: symbol = 'EUR='

In [6]: data = pd.DataFrame(raw[symbol])

In [7]: data['Доходность'] = np.log(data / data.shift(1))

In [8]: data.dropna(inplace=True)
```

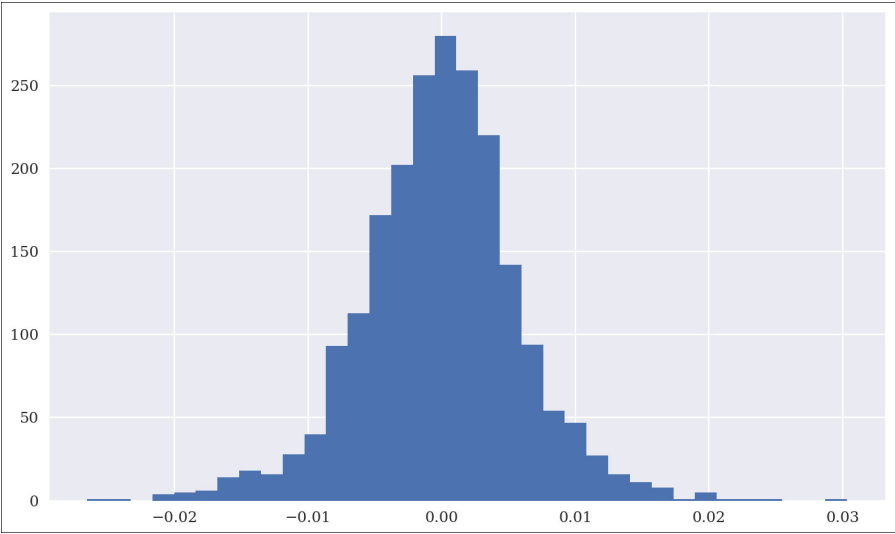
```
In [9]: data['Направление'] = np.sign(data['Доходность']).astype(int)
```

```
In [10]: data.head()
```

Out[10]:

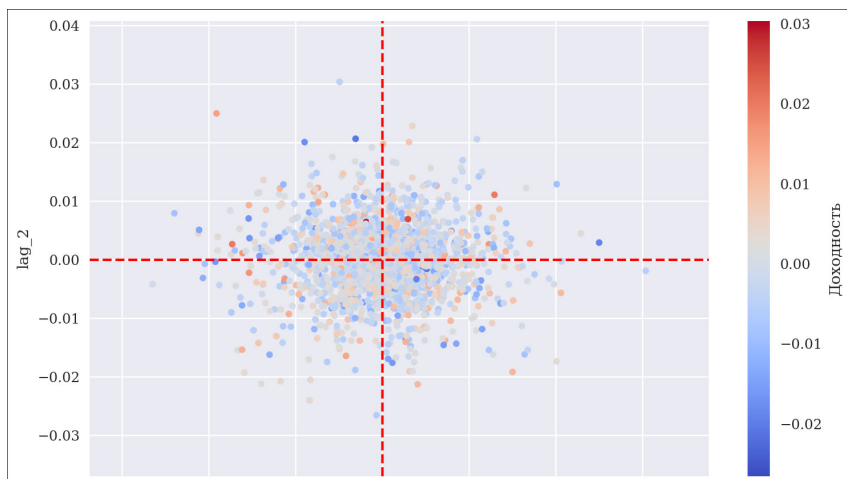
	Date	EUR=	Доходность	Направление
	2010-01-05	1.4368	-0.002988	-1
	2010-01-06	1.4412	0.003058	1
	2010-01-07	1.4318	-0.006544	-1
	2010-01-08	1.4412	0.006544	1
	2010-01-11	1.4513	0.006984	1

```
In [11]: data['Доходность'].hist(bins=35, figsize=(10, 6));
```



*Рис. 15.6. Гистограмма логарифмической доходности валютной пары EUR/USD*

Далее мы рассчитываем финансовые показатели, смещая временной ряд логарифмической доходности, и визуализируем полученный результат в виде диаграммы рассеяния (рис. 15.7).



*Рис. 15.7. Диаграмма рассеяния, основанная на значениях финансовых показателей и меток*

In [12]: lags = 2

```
In [13]: def create_lags(data):
    global cols
    cols = []
    for lag in range(1, lags + 1):
        col = 'lag_{}'.format(lag)
        data[col] = data['Доходность'].shift(lag)
        cols.append(col)
```

In [14]: create\_lags(data)

In [15]: data.head()

Out[15]:

	EUR=	Доходность	Направление	lag_1 \
Date				
2010-01-05	1.4368	-0.002988	-1	NaN
2010-01-06	1.4412	0.003058	1	-0.002988
2010-01-07	1.4318	-0.006544	-1	0.003058
2010-01-08	1.4412	0.006544	1	-0.006544
2010-01-11	1.4513	0.006984	1	0.006544

Date	lag_2
2010-01-05	NaN
2010-01-06	NaN
2010-01-07	-0.002988
2010-01-08	0.003058
2010-01-11	-0.006544

```
In [16]: data.dropna(inplace=True)
```

```
In [17]: data.plot.scatter(x='lag_1', y='lag_2', c='Доходность',
                           cmap='coolwarm', figsize=(10, 6),
                           colorbar=True)
plt.axvline(0, c='r', ls='--')
plt.axhline(0, c='r', ls='--');
```

## Регрессия

После того как набор данных подготовлен, можно выполнить регрессию по методу наименьших квадратов, чтобы выявить потенциально существующие (линейные) зависимости, спрогнозировать движение рынка на основе исследуемых показателей и протестировать торговую стратегию по этим прогнозам. Здесь возможны два подхода: представить зависимую переменную значениями *логарифмической доходности* или только значениями *направления движения*. В любом случае прогнозные значения будут вещественными, а значит, нам нужно преобразовать их в +1 или -1, чтобы анализировать только направление прогноза.

```
In [18]: from sklearn.linear_model import LinearRegression ❶
```

```
In [19]: model = LinearRegression() ❷
```

```
In [20]: data['pos_ols_1'] = model.fit(data[cols],
                                       data['Доходность']).predict(data[cols]) ❸
```

```
In [21]: data['pos_ols_2'] = model.fit(data[cols],
                                       data['Направление']).predict(data[cols]) ❹
```

```
In [22]: data[['pos_ols_1', 'pos_ols_2']].head()
```

```
Out[22]:
```

	pos_ols_1	pos_ols_2
Date		
2010-01-07	-0.000166	-0.000086

2010-01-08	0.000017	0.040404
2010-01-11	-0.000244	-0.011756
2010-01-12	-0.000139	-0.043398
2010-01-13	-0.000022	0.002237

```
In [23]: data[['pos_ols_1', 'pos_ols_2']] = np.where(
        data[['pos_ols_1', 'pos_ols_2']] > 0, 1, -1) ④
```

```
In [24]: data['pos_ols_1'].value_counts() ⑤
```

```
Out[24]: -1      1847
         1       288
         Name: pos_ols_1, dtype: int64
```

```
In [25]: data['pos_ols_2'].value_counts() ⑤
```

```
Out[25]: 1      1377
        -1       758
         Name: pos_ols_2, dtype: int64
```

```
In [26]: (data['pos_ols_1'].diff() != 0).sum() ⑥
```

```
Out[26]: 555
```

```
In [27]: (data['pos_ols_2'].diff() != 0).sum() ⑥
```

```
Out[27]: 762
```

- ① Используется реализация метода наименьших квадратов из библиотеки Scikit-learn.
- ② Регрессия применяется непосредственно к значениям *логарифмической доходности*...
- ③ ...и к значениям *направления движения*, которые представляют для нас основной интерес.
- ④ Преобразование вещественных прогнозов в категории направления (+1 и -1).
- ⑤ В целом оба подхода дают разные прогнозы по направлению движения рыночных цен.
- ⑥ Тем не менее оба они предполагают совершение относительно большого количества сделок со временем.

Имея на руках прогноз по направлению движения рынка, можно приступить к векторизованному тестированию полученных торговых стратегий на исторических данных. На этом этапе анализ строится на основе ряда упрощающих предположений, в частности о нулевых транзакционных издержках и

использовании одного и того же набора данных для обучения и тестирования. В результате обе регрессионные стратегии оказываются более эффективными по сравнению с пассивной эталонной инвестицией, но только стратегия, основанная на анализе направления рыночного движения, характеризуется общей положительной динамикой (рис. 15.8).

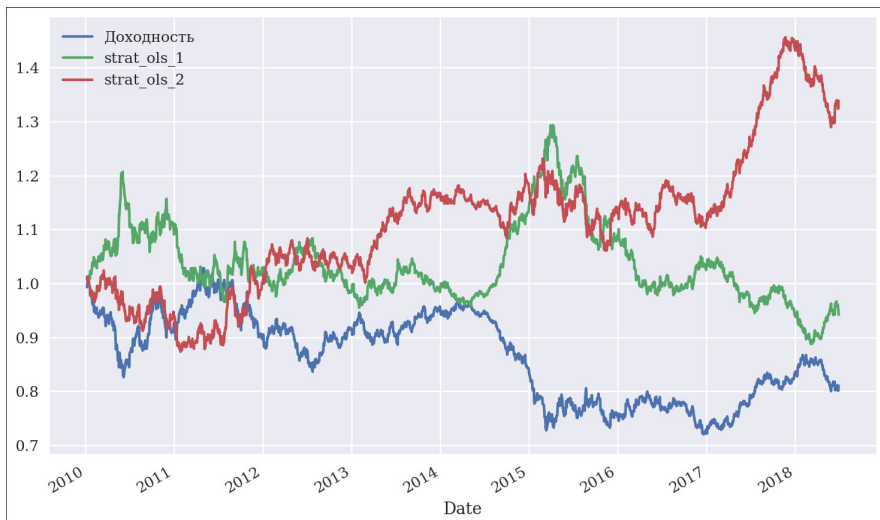


Рис. 15.8. Доходность валютной пары EUR/USD и регрессионных стратегий

```
In [28]: data['strat_ols_1'] = data['pos_ols_1'] * data['Доходность']
```

```
In [29]: data['strat_ols_2'] = data['pos_ols_2'] * data['Доходность']
```

```
In [30]: data[['Доходность', 'strat_ols_1',
               'strat_ols_2']].sum().apply(np.exp)
```

```
Out[30]: Доходность    0.810644
         strat_ols_1    0.942422
         strat_ols_2    1.339286
         dtype: float64
```

```
In [31]: (data['Направление'] == data['pos_ols_1']).value_counts() ❶
```

```
Out[31]: False    1093
         True     1042
         dtype: int64
```

```
In [32]: (data['Направление'] == data['pos_ols_2']).value_counts() ❶
```

```
Out[32]: True     1096
```

```
False    1039  
dtype: int64
```

```
In [33]: data[['Доходность', 'strat_ols_1', 'strat_ols_2']].cumsum(  
        ).apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Количество правильных и ложных прогнозов по стратегиям.

## Кластеризация

В этом разделе мы применим метод  $k$ -средних, описанный в главе 13, к финансовому временному ряду, чтобы автоматически разделить его на кластеры, используемые при формировании торговой стратегии. Идея состоит в том, что алгоритм выявляет два кластера признаков, которые предсказывают движение рынка либо вверх, либо вниз.

В следующем коде метод  $k$ -средних реализуется для двух рассмотренных ранее финансовых показателей. Результат разбивки на два кластера показан на рис. 15.9.

```
In [34]: from sklearn.cluster import KMeans
```

```
In [35]: model = KMeans(n_clusters=2, random_state=0) ❶
```

```
In [36]: model.fit(data[cols])
```

```
Out[36]: KMeans(algorithm='auto', copy_x=True, init='k-means++',  
               max_iter=300, n_clusters=2, n_init=10, n_jobs=None,  
               precompute_distances='auto', random_state=0,  
               tol=0.0001, verbose=0)
```

```
In [37]: data['pos_clus'] = model.predict(data[cols])
```

```
In [38]: data['pos_clus'] = np.where(data['pos_clus'] == 1, -1, 1) ❷
```

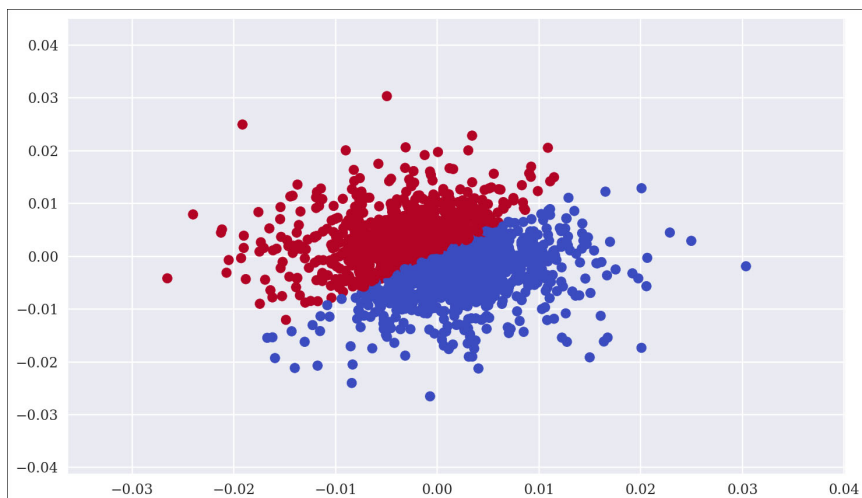
```
In [39]: data['pos_clus'].values
```

```
Out[39]: array([-1, 1, -1, ..., 1, 1, -1])
```

```
In [40]: plt.figure(figsize=(10, 6))  
        plt.scatter(data[cols].iloc[:, 0], data[cols].iloc[:, 1],  
                   c=data['pos_clus'], cmap='coolwarm');
```

- ❶ Алгоритм разбивает анализируемые данные на два кластера.  
❷ Направление движения рынка определяется по значениям кластера.





*Рис. 15.9. Два кластера, выявленных по методу  $k$ -средних*

Вполне очевидно, что разбивка на кластеры реализуется произвольным образом. В конце концов, откуда алгоритм знает, что именно нужно искать? Тем не менее полученная торговая стратегия демонстрирует незначительное преимущество по сравнению с пассивной эталонной инвестицией (рис. 15.10). Примечательно, что алгоритм работает в отсутствие явно заданных признаков (без учителя), а коэффициент попадания — количество правильных предсказаний среди всех сделанных — составляет менее 50%.

```
In [41]: data['strat_clus'] = data['pos_clus'] * data['Доходность']
```

```
In [42]: data[['Доходность', 'strat_clus']].sum().apply(np.exp)
```

```
Out[42]: Доходность      0.810644
strat_clus      1.277133
dtype: float64
```

```
In [43]: (data['Направление'] == data['pos_clus']).value_counts()
```

```
Out[43]: True      1077
False    1058
dtype: int64
```

```
In [44]: data[['Доходность', 'strat_clus']].cumsum(
          ).apply(np.exp).plot(figsize=(10, 6));
```



Рис. 15.10. Доходность валютной пары EUR/USD и торговой стратегии, основанной на методе *k-средних*

## Частотный подход

Можно пойти более простым путем и реализовать *частотный подход* к прогнозированию направления движения финансовых рынков. Для этого необходимо преобразовать два вещественных признака в бинарные и оценить вероятность движения вверх или вниз на основе исторических наблюдений с учетом четырех возможных комбинаций признаков: (0, 0), (0, 1), (1, 0), (1, 1).

Такой подход относительно легко реализовать благодаря встроенным функциям анализа данных библиотеки *pandas*.

```
In [45]: def create_bins(data, bins=[0]):
         global cols_bin
         cols_bin = []
         for col in cols:
             col_bin = col + '_bin'
             data[col_bin] = np.digitize(data[col], bins=bins) ❶
             cols_bin.append(col_bin)
```

```
In [46]: create_bins(data)
```

```
In [47]: data[cols_bin + ['Направление']].head() ❷
```

Out[47]:

	lag_1_bin	lag_2_bin	Направление
Date			
2010-01-07	1	0	-1
2010-01-08	0	1	1
2010-01-11	1	0	1
2010-01-12	1	1	-1
2010-01-13	0	1	1

In [48]: grouped = data.groupby(cols\_bin + ['Направление'])  
grouped.size() ③

Out[48]:

lag_1_bin	lag_2_bin	Направление	
0	0	-1	239
		0	4
		1	258
1	1	-1	262
		1	288
		-1	272
	0	0	1
		1	278
		-1	278
	1	0	4
		1	251

dtype: int64

In [49]: res = grouped['Направление'].size().unstack(fill\_value=0) ④

In [50]: def highlight\_max(s):  
is\_max = s == s.max()  
return ['background-color: yellow' if v else ''  
for v in is\_max] ⑤

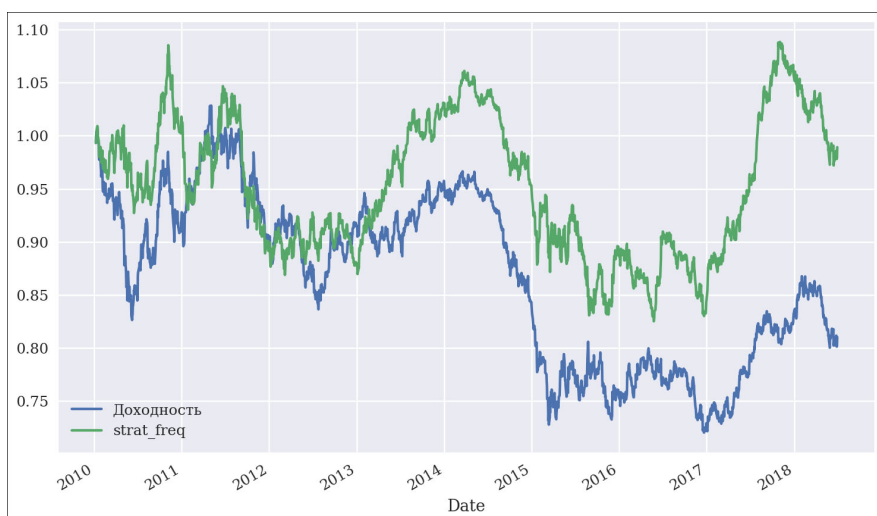
In [51]: res.style.apply(highlight\_max, axis=1) ⑤

Out[51]:

	Направление	-1	0	1
lag_1_bin	lag_2_bin			
0	0	239	4	258
		1	262	0
		0	272	1
1	1	278	4	251

- ❶ Дискретизация значений признаков с учетом параметра bins.
- ❷ Отображение дискретных значений признаков и меток.
- ❸ Отображение частоты возможных движений в зависимости от комбинаций значений признаков.
- ❹ Преобразование объекта DataFrame, чтобы частоты хранились по столбцам.
- ❺ Определение наиболее часто встречающегося значения для каждой комбинации значений признаков.

Три комбинации значений признаков указывают на большую вероятность движения вниз, и только для одной из них более вероятным оказывается движение вверх. На основе этого можно построить торговую стратегию, график доходности которой показан на рис. 15.11.



*Рис. 15.11. Доходность валютной пары EUR/USD и торговой стратегии, основанной на частотном подходе*

```
In [52]: data['pos_freq'] = np.where(data[cols_bin].sum(axis=1) ==
                                     2, -1, 1) ❶
```

```
In [53]: (data['Направление'] == data['pos_freq']).value_counts()
Out[53]: True      1102
         False    1033
         dtype: int64
```

```
In [54]: data['strat_freq'] = data['pos_freq'] * data['Доходность']
```

```
In [55]: data[['Доходность', 'strat_freq']].sum().apply(np.exp)
```

```
Out[55]: Доходность      0.810644  
         strat_freq      0.989513  
         dtype: float64
```

```
In [56]: data[['Доходность', 'strat_freq']].cumsum(  
          ).apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Преобразование полученного частотного распределения в торговую стратегию.

## Классификация

В этом разделе мы воспользуемся алгоритмами машинного обучения (см. главу 13) для решения задачи о прогнозировании направления движения цен на финансовых рынках. С учетом примеров из предыдущих разделов применить логистическую регрессию, наивный байесовский классификатор и метод опорных векторов будет не намного сложнее, чем в случае обработки небольшого набора выборочных данных.

### Два бинарных признака

Для начала напомним код обучения моделей на основе двух бинарных признаков и получения соответствующих значений позиций.

```
In [57]: from sklearn import linear_model  
         from sklearn.naive_bayes import GaussianNB  
         from sklearn.svm import SVC
```

```
In [58]: C = 1
```

```
In [59]: models = {  
          'log_reg': linear_model.LogisticRegression(C=C),  
          'gauss_nb': GaussianNB(),  
          'svm': SVC(C=C)  
        }
```

```
In [60]: def fit_models(data): ❶
         mfit = {model: models[model].fit(data[cols_bin],
                                           data['Направление'])
                 for model in models.keys()}
```

```
In [61]: fit_models(data)
```

```
In [62]: def derive_positions(data): ❷
         for model in models.keys():
             data['pos_' + model] = models[model].
                 predict(data[cols_bin])
```

```
In [63]: derive_positions(data)
```

❶ Функция, отвечающая за обучение всех моделей.

❷ Функция, определяющая значения всех позиций для обученных моделей.

Теперь можно переходить к векторизованному тестированию полученных торговых стратегий. Графики их доходностей приведены на рис. 15.12.

```
In [64]: def evaluate_strats(data): ❶
         global sel
         sel = []
         for model in models.keys():
             col = 'strat_' + model
             data[col] = data['pos_' + model] * data['Доходность']
             sel.append(col)
         sel.insert(0, 'Доходность')
```

```
In [65]: evaluate_strats(data)
```

```
In [66]: sel.insert(1, 'strat_freq')
```

```
In [67]: data[sel].sum().apply(np.exp) ❷
```

```
Out[67]: Доходность      0.810644
         strat_freq      0.989513
         strat_log_reg    1.243322
         strat_gauss_nb    1.243322
         strat_svm        0.989513
         dtype: float64
```

```
In [68]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Функция оценки всех торговых стратегий.
- ❷ Некоторые стратегии приводят к одинаковым результатам.

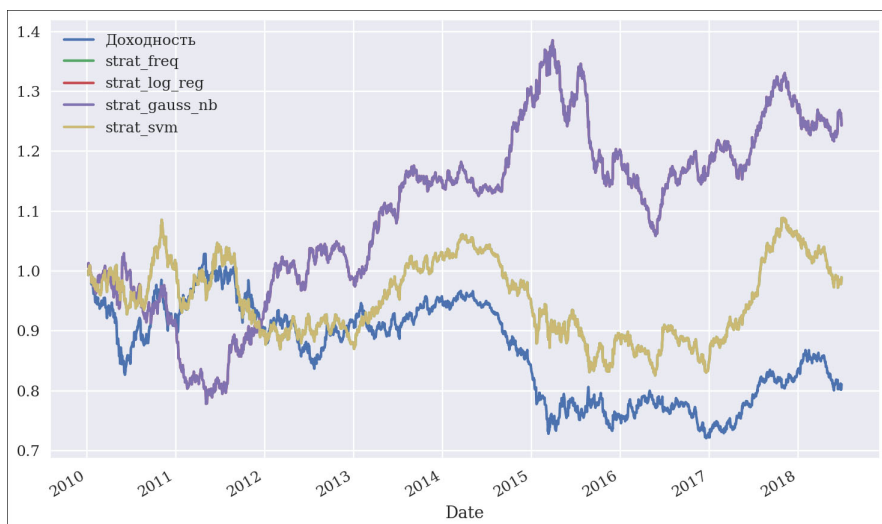


Рис. 15.12. Доходность валютной пары EUR/USD и торговых стратегий на основе классификации (по двум бинарным признакам)

## Пять бинарных признаков

В попытке повысить доходность разрабатываемых стратегий в следующем коде количество учитываемых бинарных признаков увеличивается с двух до пяти. Это, в частности, позволяет значительно улучшить доходность стратегии на основе метода опорных векторов (рис. 15.13). С другой стороны, доходность стратегий на основе логистической регрессии и наивного байесовского классификатора ухудшается.

```
In [69]: data = pd.DataFrame(raw[symbol])
In [70]: data['Доходность'] = np.log(data / data.shift(1))

In [71]: data['Направление'] = np.sign(data['Доходность'])

In [72]: lags = 5 ❶
         create_lags(data)
         data.dropna(inplace=True)
```

```
In [73]: create_bins(data) ❷
```

```
cols_bin
```

```
Out[73]: ['lag_1_bin', 'lag_2_bin', 'lag_3_bin', 'lag_4_bin',  
         'lag_5_bin']
```

```
In [74]: data[cols_bin].head()
```

```
Out[74]:
```

	lag_1_bin	lag_2_bin	lag_3_bin	lag_4_bin	\
Date					
2010-01-12	1	1	0	1	
2010-01-13	0	1	1	0	
2010-01-14	1	0	1	1	
2010-01-15	0	1	0	1	
2010-01-19	0	0	1	0	

	lag_5_bin
Date	
2010-01-12	0
2010-01-13	1
2010-01-14	0
2010-01-15	1
2010-01-19	1

```
In [75]: data.dropna(inplace=True)
```

```
In [76]: fit_models(data)
```

```
In [77]: derive_positions(data)
```

```
In [78]: evaluate_strats(data)
```

```
In [79]: data[sel].sum().apply(np.exp)
```

```
Out[79]: Доходность      0.805002  
strat_log_reg    0.971623  
strat_gauss_nb   0.986420  
strat_svm        1.452406  
dtype: float64
```

```
In [80]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Теперь создаются пять временных рядов логарифмической доходности.
- ❷ Преобразование вещественных значений признаков в бинарные категории.





Рис. 15.13. Доходность валютной пары EUR/USD и торговых стратегий на основе классификации (по пяти бинарным признакам)

## Пять дискретизированных признаков

Наконец, в следующем коде признаки дискретизируются по первому и второму моментам распределения исторических доходностей, что позволяет получить больше возможных комбинаций значений признаков. Это повышает эффективность всех алгоритмов классификации, но, как и ранее, наибольшее увеличение доходности наблюдается в методе опорных векторов (рис. 15.14).

```
In [81]: mu = data['Доходность'].mean() ❶
         v = data['Доходность'].std()    ❷

In [82]: bins = [mu - v, mu, mu + v]    ❸
         bins ❸

Out[82]: [-0.006033537040418665, -0.00010174015279231306,
          0.005830056734834039]

In [83]: create_bins(data, bins)

In [84]: data[cols_bin].head()
Out[84]:
```

	lag_1_bin	lag_2_bin	lag_3_bin	lag_4_bin \
Date				
2010-01-12	3	3	0	2

2010-01-13	1	3	3	0
2010-01-14	2	1	3	3
2010-01-15	1	2	1	3
2010-01-19	0	1	2	1

	lag_5_bin
Date	
2010-01-12	1
2010-01-13	2
2010-01-14	0
2010-01-15	3
2010-01-19	3

```
In [85]: fit_models(data)
```

```
In [86]: derive_positions(data)
```

```
In [87]: evaluate_strats(data)
```

```
In [88]: data[sel].sum().apply(np.exp)
```

```
Out[88]: Доходность      0.805002
strat_log_reg    1.431120
strat_gauss_nb   1.815304
strat_svm        5.653433
dtype: float64
```

```
In [89]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

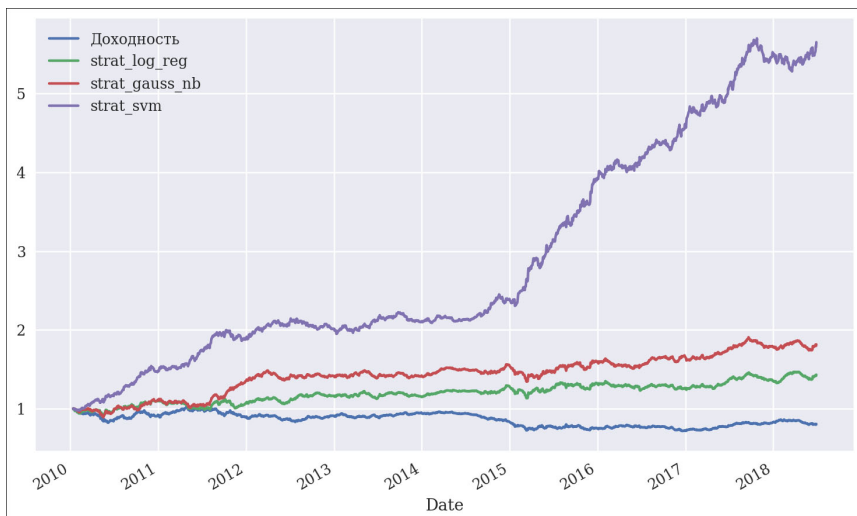
- ❶ Показатели среднего...
- ❷ ...и стандартного отклонения...
- ❸ ...используются для дискретизации признаков.



### Типы признаков

В этой главе в качестве значений признаков используются данные о доходности (обычно приведенные к бинарному или дискретизированному формату), дублируемые с временным смещением. Это в основном сделано для удобства, поскольку значения признаков можно получить непосредственно из финансового временного ряда. Но на практике такие значения могут поступать из самых разных источников и включать другие временные ряды, статистические показатели, макроэкономические данные, финансовые

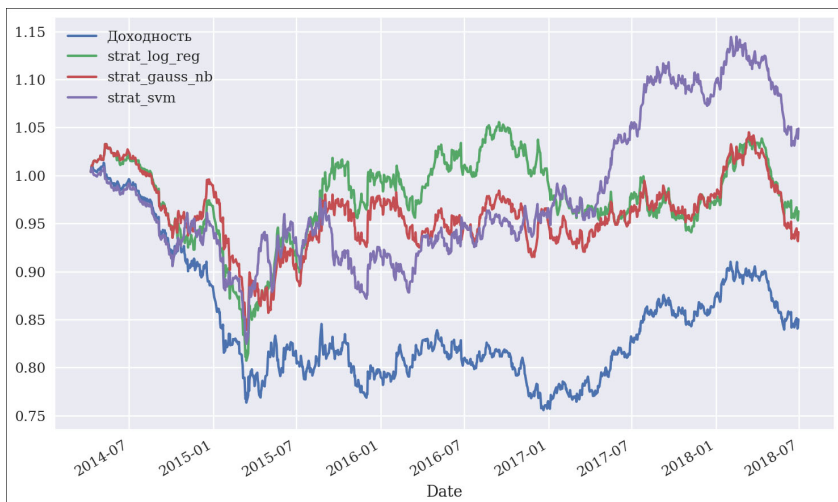
индикаторы компаний и т.п. Подробнее эта тема рассмотрена в книге Лопеса де Прадо [5]. Существуют специализированные пакеты Python, предназначенные для автоматизированного извлечения признаков временных рядов, например `tsfresh` (<https://github.com/blue-yonder/tsfresh>).



*Рис. 15.14. Доходность валютной пары EUR/USD и торговых стратегий на основе классификации (по пяти дискретизированным признакам)*

## Последовательное разделение данных на обучающий и тестовый наборы

Чтобы лучше оценить эффективность алгоритмов классификации, мы реализуем принцип *последовательного* разделения данных на обучающий и тестовый наборы. Идея заключается в том, чтобы смоделировать ситуацию, когда для тренировки алгоритма машинного обучения доступны только данные до определенного момента времени. Соответственно, когда начнутся реальные торги, программа столкнется с неизвестными для себя данными. Именно так проверяется эффективность алгоритма. В данном конкретном случае все алгоритмы классификации (при тех же упрощающих предположениях, что и раньше) показывают лучшие результаты по сравнению с пассивной эталонной инвестицией, но положительная динамика роста абсолютной доходности наблюдается только у метода линейной регрессии и наивного байесовского классификатора (рис. 15.15).



*Рис. 15.15. Доходность валютной пары EUR/USD и торговых стратегий на основе классификации (последовательное разделение данных на обучающий и тестовый наборы)*

```
In [90]: split = int(len(data) * 0.5)
```

```
In [91]: train = data.iloc[:split].copy() ❶
```

```
In [92]: fit_models(train) ❶
```

```
In [93]: test = data.iloc[split:].copy() ❷
```

```
In [94]: derive_positions(test) ❷
```

```
In [95]: evaluate_strats(test) ❷
```

```
In [96]: test[sel].sum().apply(np.exp)
```

```
Out[96]: Доходность      0.850291
         strat_log_reg   0.962989
         strat_gauss_nb  0.941172
         strat_svm       1.048966
         dtype: float64
```

```
In [97]: test[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Тренировка всех алгоритмов классификации на обучающем наборе.
- ❷ Проверка всех алгоритмов классификации на тестовом наборе.

## Рандомизированное разделение данных на обучающий и тестовый наборы

Алгоритмы классификации обучаются и тестируются на бинарных или дискретизированных признаках. Идея заключается в том, что выявляемые шаблоны (паттерны) признаков позволяют предсказывать будущие движения рынка с точностью, превышающей 50%. При этом неявно предполагается, что прогнозная эффективность шаблонов сохраняется со временем. В таком случае не имеет особого значения, на каком наборе данных обучается или тестируется алгоритм, если, конечно, временной ряд можно разбить на обучающий и тестовый наборы.

Обычно эффективность алгоритма классификации проверяется на данных вне выборки путем *рандомизированного* разделения данных на обучающий и тестовый наборы. Это, опять-таки, позволяет смоделировать реальную ситуацию, когда во время торгов программа начинает постоянно сталкиваться с неизвестными данными. Применяемый здесь подход аналогичен тому, который рассматривался в главе 13. В результате метод опорных векторов снова оказывается наиболее эффективным при работе с данными вне выборки (рис. 15.16).

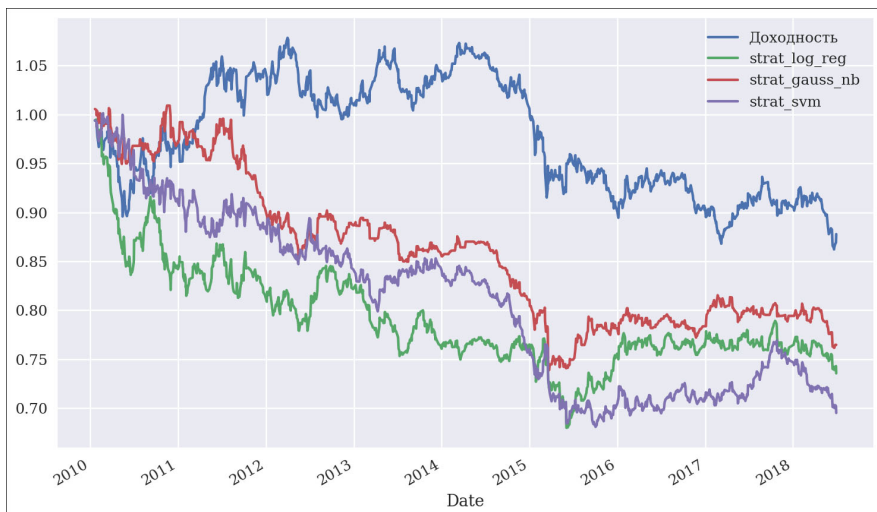


Рис. 15.16. Доходность валютной пары EUR/USD и торговых стратегий на основе классификации (рандомизированное разделение данных на обучающий и тестовый наборы)

```
In [98]: from sklearn.model_selection import train_test_split
```

```
In [99]: train, test = train_test_split(data, test_size=0.5,  
                                         shuffle=True,  
                                         random_state=100)
```

```
In [100]: train = train.copy().sort_index() ❶
```

```
In [101]: train[cols_bin].head()
```

```
Out[101]:
```

	lag_1_bin	lag_2_bin	lag_3_bin	lag_4_bin \
Date				
2010-01-12	3	3	0	2
2010-01-13	1	3	3	0
2010-01-14	2	1	3	3
2010-01-15	1	2	1	3
2010-01-20	1	0	1	2

	lag_5_bin
Date	
2010-01-12	1
2010-01-13	2
2010-01-14	0
2010-01-15	3
2010-01-20	1

```
In [102]: test = test.copy().sort_index() ❶
```

```
In [103]: fit_models(train)
```

```
In [104]: derive_positions(test)
```

```
In [105]: evaluate_strats(test)
```

```
In [106]: test[sel].sum().apply(np.exp)
```

```
Out[106]: Доходность      0.878078  
          strat_log_reg   0.735893  
          strat_gauss_nb  0.765009  
          strat_svm       0.695428  
          dtype: float64
```

```
In [107]: test[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Обучающий и тестовый наборы сортируются в хронологическом порядке.

## Глубокие нейронные сети

Глубокие нейронные сети (Deep Neural Networks — DNN) пытаются моделировать работу человеческого мозга. Они обычно состоят из входного слоя (признаков), выходного слоя (метки) и определенного количества скрытых слоев. Наличие скрытых слоев как раз и делает нейронную сеть *глубокой*. Такие нейронные сети способны обучаться сложным зависимостям и демонстрировать более высокую эффективность при решении целого ряда задач. В этом контексте обычно говорят не о машинном обучении, а о *глубоком обучении*. Введением в данную тему могут послужить книги Жерона [8], а также Паттерсона и Гибсона [6].

### DNN и библиотека Scikit-learn

В этом разделе мы применим алгоритм `MLPClassifier` из библиотеки `Scikit-learn`, с которым познакомились в главе 13. Сначала он обучается и тестируется на всем наборе данных с использованием дискретизированных признаков. Алгоритм демонстрирует исключительную доходность в пределах выборки (рис. 15.17), что свидетельствует об огромных возможностях DNN в решении такого рода задач. Но вероятнее всего, мы имеем дело с сильным переобучением, поскольку доходность выглядит неестественно высокой.

```
In [108]: from sklearn.neural_network import MLPClassifier
```

```
In [109]: model = MLPClassifier(solver='lbfgs', alpha=1e-5,  
                                hidden_layer_sizes=2 * [250],  
                                random_state=1)
```

```
In [110]: %time model.fit(data[cols_bin], data['Направление'])  
CPU times: user 16.1 s, sys: 156 ms, total: 16.2 s  
Wall time: 9.85 s
```

```
Out[110]: MLPClassifier(activation='relu', alpha=1e-05,  
                          batch_size='auto', beta_1=0.9, beta_2=0.999,  
                          early_stopping=False, epsilon=1e-08,  
                          hidden_layer_sizes=[250, 250],  
                          learning_rate='constant',  
                          learning_rate_init=0.001, max_iter=200,  
                          momentum=0.9, n_iter_no_change=10,  
                          nesterovs_momentum=True, power_t=0.5,  
                          random_state=1, shuffle=True, solver='lbfgs',
```

```
tol=0.0001, validation_fraction=0.1,  
verbose=False, warm_start=False)
```

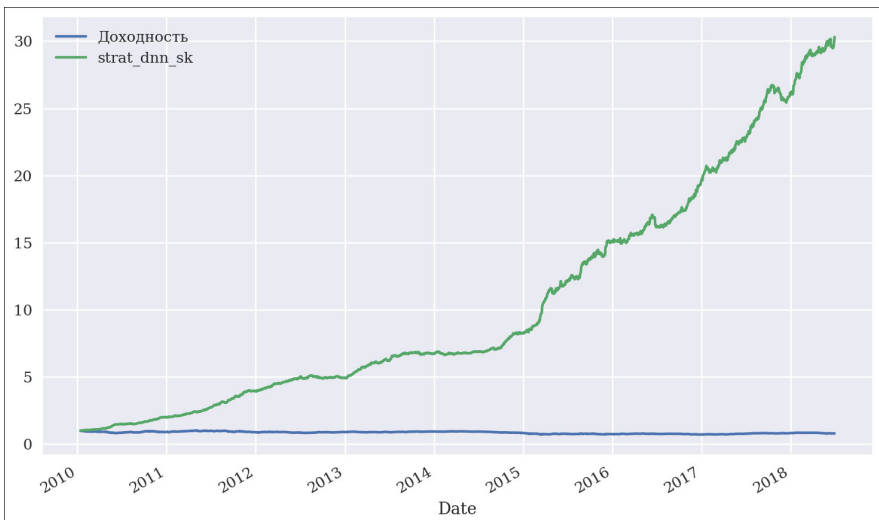
```
In [111]: data['pos_dnn_sk'] = model.predict(data[cols_bin])
```

```
In [112]: data['strat_dnn_sk'] = data['pos_dnn_sk'] * data['Доходность']
```

```
In [113]: data[['Доходность', 'strat_dnn_sk']].sum().apply(np.exp)
```

```
Out[113]: Доходность      0.805002  
          strat_dnn_sk   35.156677  
          dtype: float64
```

```
In [114]: data[['Доходность', 'strat_dnn_sk']].cumsum().apply(  
          np.exp).plot(figsize=(10, 6));
```



*Рис. 15.17. Доходность валютной пары EUR/USD и торговой стратегии на основе глубокой нейронной сети (библиотека Scikit-learn, данные в пределах выборки)*

Чтобы избежать переобучения модели, на следующем этапе мы будем случайным образом разделять исходные данные на обучающий и тестовый наборы. Как и ранее, такой алгоритм показывает более высокую доходность, чем пассивная эталонная инвестиция (рис. 15.18), а кроме того, он демонстрирует положительную динамику роста абсолютной доходности. Зато теперь полученный результат выглядит более реалистично.



```

In [115]: train, test = train_test_split(data, test_size=0.5,
                                         random_state=100)

In [116]: train = train.copy().sort_index()

In [117]: test = test.copy().sort_index()

In [118]: model = MLPClassifier(solver='lbfgs', alpha=1e-5,
                               max_iter=500,
                               hidden_layer_sizes=3 * [500],
                               random_state=1) ❶

In [119]: %time model.fit(train[cols_bin], train['Направление'])
CPU times: user 2min 26s, sys: 1.02 s, total: 2min 27s
Wall time: 1min 31s
Out[119]: MLPClassifier(activation='relu', alpha=1e-05,
                        batch_size='auto', beta_1=0.9, beta_2=0.999,
                        early_stopping=False, epsilon=1e-08,
                        hidden_layer_sizes=[500, 500, 500],
                        learning_rate='constant',
                        learning_rate_init=0.001, max_iter=500,
                        momentum=0.9, n_iter_no_change=10,
                        nesterovs_momentum=True, power_t=0.5,
                        random_state=1, shuffle=True, solver='lbfgs',
                        tol=0.0001, validation_fraction=0.1,
                        verbose=False, warm_start=False)

In [120]: test['pos_dnn_sk'] = model.predict(test[cols_bin])

In [121]: test['strat_dnn_sk'] = test['pos_dnn_sk'] * test['Доходность']

In [122]: test[['Доходность', 'strat_dnn_sk']].sum().apply(np.exp)
Out[122]: Доходность      0.878078
          strat_dnn_sk    1.242042
          dtype: float64

In [123]: test[['Доходность', 'strat_dnn_sk']].cumsum(
          ).apply(np.exp).plot(figsize=(10, 6));

```

❶ Увеличение количества скрытых слоев и нейронов.



*Рис. 15.18. Доходность валютной пары EUR/USD и торговой стратегии на основе глубокой нейронной сети (библиотека Scikit-learn, рандомизированное разделение данных на обучающий и тестовый наборы)*

## DNN и библиотека TensorFlow

TensorFlow — популярная платформа глубокого обучения. Она разработана компанией Google и применяется для решения широкого круга задач машинного обучения<sup>5</sup>.

Поскольку мы уже познакомились с TensorFlow в главе 13, будет несложно применить алгоритм `DNNClassifier` для формирования торговой стратегии. Обучающий и тестовый наборы здесь такие же, как и раньше. Сначала выполняется тренировка модели. В пределах выборки алгоритм оказывается эффективнее пассивной эталонной инвестиции, показывая высокую абсолютную доходность (рис. 15.19), что свидетельствует о возможном переобучении.

```
In [124]: import tensorflow as tf
          tf.logging.set_verbosity(tf.logging.ERROR)
```

```
In [125]: fc = [tf.contrib.layers.real_valued_column('lags',
          dimension=lags)]
```

<sup>5</sup> О применении TensorFlow в глубоком обучении можно прочитать в книге Задеха и Рамсундара [7].

```

In [126]: model = tf.contrib.learn.DNNClassifier(hidden_units=
          3 * [500], n_classes=len(bins) + 1,
          feature_columns=fc)

In [127]: def input_fn():
          fc = {'lags': tf.constant(data[cols_bin].values)}
          la = tf.constant(data['Направление'].apply(
              lambda x: 0 if x < 0 else 1).values,
              shape=[data['Направление'].size, 1])
          return fc, la

In [128]: %time model.fit(input_fn=input_fn, steps=250) ❶
CPU times: user 2min 7s, sys: 8.85 s, total: 2min 16s
Wall time: 49 s

Out[128]: DNNClassifier(params={'head':
<tensorflow.contrib.learn.python.learn.estimators.head.
_MultiClassHead object at 0x1a19acf898>, 'hidden_units':
[500, 500, 500], 'feature_columns':
(_RealValuedColumn(column_name='lags', dimension=5,
default_value=None, dtype=tf.float32, normalizer=None)),
'optimizer': None, 'activation_fn': <function relu at
0x1161441e0>, 'dropout': None, 'gradient_clip_norm': None,
'embedding_lr_multipliers': None,
'input_layer_min_slice_size': None})

In [129]: model.evaluate(input_fn=input_fn, steps=1) ❷
Out[129]: {'loss': 0.6879357, 'accuracy': 0.5379925,
          'global_step': 250}

In [130]: pred = np.array(list(model.predict(input_fn=input_fn))) ❷
          pred[:10] ❷
Out[130]: array([0, 0, 0, 0, 0, 1, 0, 1, 1, 0])

In [131]: data['pos_dnn_tf'] = np.where(pred > 0, 1, -1) ❸

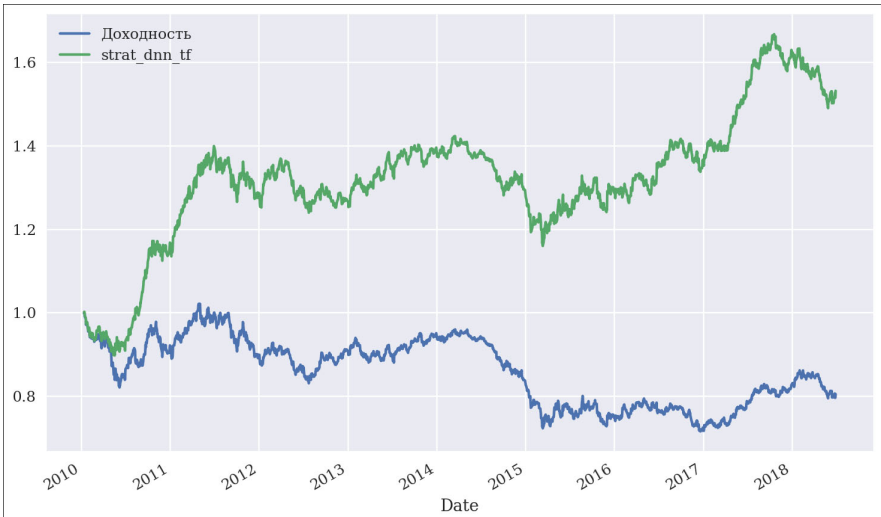
In [132]: data['strat_dnn_tf'] = data['pos_dnn_tf'] * data['Доходность']

In [133]: data[['Доходность', 'strat_dnn_tf']].sum().apply(np.exp)
Out[133]: Доходность      0.805002
          strat_dnn_tf    2.437222
          dtype: float64

```

```
In [134]: data[['Доходность', 'strat_dnn_tf']].cumsum(
          ).apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Обучение модели может занять какое-то время.
- ❷ Бинарные прогнозы (0, 1) ...
- ❸ ...необходимо преобразовать в рыночные позиции (-1, +1).



*Рис. 15.19. Доходность валютной пары EUR/USD и торговой стратегии на основе глубокой нейронной сети (библиотека TensorFlow, данные в пределах выборки)*

Ниже снова реализован рандомизированный подход к разделению обучающего и тестового наборов, что позволяет получить более реалистичное представление об эффективности алгоритмической торговой стратегии на основе DNN. Как и следовало ожидать, доходность вне выборки оказывается ниже (рис. 15.20). Кроме того, из-за специфической параметризации алгоритм DNNClassifier библиотеки TensorFlow ощутимо уступает алгоритму MLPClassifier библиотеки Scikit-learn.

```
In [135]: model = tf.contrib.learn.DNNClassifier(hidden_units=
          3 * [500], n_classes=len(bins) + 1,
          feature_columns=fc)
```

```
In [136]: data = train
```

```

In [137]: %time model.fit(input_fn=input_fn, steps=2500)
CPU times: user 11min 7s, sys: 1min 7s, total: 12min 15s
Wall time: 4min 27s
Out[137]: DNNClassifier(params={'head':
    <tensorflow.contrib.learn.python.learn.estimators.head.
    _MultiClassHead object at 0x116828cc0>, 'hidden_units':
    [500, 500, 500], 'feature_columns':
    (_RealValuedColumn(column_name='lags', dimension=5,
    default_value=None, dtype=tf.float32, normalizer=None)),
    'optimizer': None, 'activation_fn': <function relu at
    0x1161441e0>, 'dropout': None, 'gradient_clip_norm': None,
    'embedding_lr_multipliers': None,
    'input_layer_min_slice_size': None})

In [138]: data = test

In [139]: model.evaluate(input_fn=input_fn, steps=1)
Out[139]: {'loss': 0.82882184, 'accuracy': 0.48968107,
    'global_step': 2500}

In [140]: pred = np.array(list(model.predict(input_fn=input_fn)))

In [141]: test['pos_dnn_tf'] = np.where(pred > 0, 1, -1)

In [142]: test['strat_dnn_tf'] = test['pos_dnn_tf'] * test['Доходность']

In [143]: test[['Доходность', 'strat_dnn_sk',
    'strat_dnn_tf']].sum().apply(np.exp)
Out[143]: Доходность      0.878078
    strat_dnn_sk      1.242042
    strat_dnn_tf      1.063968
    dtype: float64

In [144]: test[['Доходность', 'strat_dnn_sk', 'strat_dnn_tf']].cumsum(
    ).apply(np.exp).plot(figsize=(10, 6));

```



*Рис. 15.20. Доходность валютной пары EUR/USD и торговой стратегии на основе глубокой нейронной сети (библиотека TensorFlow, рандомизированное разделение данных на обучающий и тестовый наборы)*



### Эффективность алгоритмов

Все результаты векторизованного тестирования на исторических данных, показанные различными алгоритмическими торговыми стратегиями, носят чисто иллюстративный характер. Помимо упрощающего предположения об отсутствии транзакционных издержек, на результаты оказывает влияние ряд других параметров (в основном выбираемых произвольным образом). Кроме того, результаты зависят от сравнительно небольшого набора данных о валютном курсе EUR/USD. Цель главы заключалась в том, чтобы продемонстрировать применение различных статистических методов и алгоритмов машинного обучения к финансовым данным, а не в том, чтобы разработать надежную алгоритмическую торговую стратегию, которую можно внедрить на практике. О практической стороне вопроса мы поговорим в следующей главе.

## Резюме

Эта глава была посвящена алгоритмическим торговым стратегиям и оценке их эффективности по результатам векторизованного тестирования на исторических данных. Вначале мы рассмотрели достаточно простую стратегию, основанную на двух скользящих средних. Такой тип стратегий известен уже достаточно давно и применяется на протяжении десятилетий. На примере этой стратегии мы познакомились с принципами векторизованного тестирования на исторических данных, а также соответствующими инструментами анализа данных, доступными в библиотеках NumPy и pandas.

Также мы изучили регрессию по методу наименьших квадратов и проверили гипотезу случайного блуждания на реальном финансовом временном ряде. Это своеобразное мерило, по которому оцениваются любые алгоритмические торговые стратегии.

Основное внимание в главе уделялось алгоритмам машинного обучения, с которыми мы впервые познакомились в главе 13. В основном это алгоритмы классификации, которые применяются по одинаковой схеме. Признаками в них служат смещенные временные ряды логарифмической доходности, хотя это не единственно возможный подход. Мы поступили так в основном из соображений удобства и простоты. Кроме того, наш анализ базировался на ряде упрощающих предположений, поскольку мы хотели сконцентрироваться на технических аспектах применения алгоритмов машинного обучения к финансовым временным рядам для прогнозирования направления рыночных движений.

## Дополнительные ресурсы

В главе упоминались следующие источники.

1. Baxter, Martin, and Andrew Rennie. *Financial Calculus, 17th Edition* (1996, Cambridge University Press).
2. Brock, William, Josef Lakonishok, and Blake LeBaron. *Simple Technical Trading Rules and the Stochastic Properties of Stock Returns* (1992, *The Journal of Finance*, Vol. 47, No. 5, pp. 1731–1764).
3. Fama, Eugene. *Random Walks in Stock Market Prices* (1965, Selected Papers, No. 16, Graduate School of Business, University of Chicago).
4. Jensen, Michael. *Some Anomalous Evidence Regarding Market Efficiency* (1978, *Journal of Financial Economics*, Vol. 6, No. 2/3, pp. 95–101).

5. Lopez de Prado, Marcos. *Advances in Financial Machine Learning* (2018, Wiley).
6. Patterson, Josh, and Adam Gibson. *Deep Learning* (2017, O'Reilly).
7. Zadeh, Reza Bosagh, and Bharath Ramsundar. *TensorFlow for Deep Learning* (2018, O'Reilly).
8. Жерон, Орельен. *Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow: концепции, инструменты и техники для создания интеллектуальных систем, 2-е издание* (пер. с англ., ООО “Диалектика”, 2021).

Рассмотренные в главе финансовые темы детально изложены в следующих книгах.

- Chan, Ernest. *Quantitative Trading* (2009, Wiley).
- Chan, Ernest. *Algorithmic Trading* (2013, Wiley).
- Chan, Ernest. *Machine Trading* (2017, Wiley).

При изучении этой главы вам также могут пригодиться следующие книги.

- Albon, Chris. *Machine Learning with Python Cookbook* (2018, O'Reilly).
- VanderPlas, Jake. *Python Data Science Handbook* (2016, O'Reilly).

Рекомендую также пройти сертификационный онлайн-курс, посвященный алгоритмической торговле на Python. Он доступен на сайте <http://certificate.tpq.io>.





---

## Автоматизированная торговля

Люди беспокоятся о том, что компьютеры станут слишком умными и захватят весь мир, но настоящая проблема в том, что они слишком глупы и уже завладели миром.

*Педро Домингос*

Итак, что дальше? В нашем распоряжении есть торговая платформа, которая позволяет извлекать исторические и потоковые финансовые данные, размещать заявки и проверять состояние счета. Мы исследовали несколько методик разработки алгоритмических торговых стратегий, основанных на прогнозировании направления движения рыночных цен. Как все это объединить для получения автоматизированной торговой системы? Универсального ответа на такой вопрос не существует. Тем не менее в данной главе мы рассмотрим ряд тем, которые помогут на него ответить. Предполагается, что автоматизированная торговля проводится согласно единственной алгоритмической торговой стратегии. Это существенно упрощает общую задачу, особенно в отношении управления капиталом и рисками.

В главе рассматриваются следующие темы.

### *Управление капиталом*

Как будет показано в этом разделе, объем сделок определяется в соответствии с критерием Келли и зависит от параметров выбранной торговой стратегии и доступных средств.

### *Торговая стратегия, основанная на машинном обучении*

Чтобы убедиться в надежности алгоритмической торговой стратегии, ее необходимо тщательно протестировать на исторических данных как в плане производительности, так и в плане риска. Рассматриваемая в этом разделе стратегия основана на алгоритме классификации, который описывался в главе 15.

### *Веб-алгоритм*

Для развертывания алгоритмической торговой стратегии в автоматизированном режиме ее нужно реализовать в виде веб-алгоритма, который

позволяет обрабатывать входящие потоковые данные в реальном времени.

### *Инфраструктура и развертывание*

Для обеспечения надежной и устойчивой работы автоматизированные стратегии алгоритмической торговли лучше всего развертывать в облаке. Это выгодно с точки зрения доступности, производительности и безопасности.

### *Протоколирование и мониторинг*

Чтобы иметь возможность отслеживать события, возникающие в процессе развертывания автоматизированной торговой стратегии, необходимо включить режим протоколирования. Постоянный мониторинг через сетевое подключение позволяет дистанционно регистрировать все важные события в реальном времени.

## Управление капиталом

Главный вопрос, на который приходится отвечать, занимаясь алгоритмической торговлей: какую долю имеющихся средств следует задействовать в выбранной торговой стратегии? Ответ на него зависит от цели, которая преследуется при развертывании такой стратегии. Большинство специалистов сходятся на том, что наиболее оправданной целью должна стать *долгосрочная максимизация капитала*. Это именно то, что имел в виду Эдвард Торп, когда ввел понятие *критерия Келли* для инвестиций [1].

### Критерий Келли в биномиальной модели

Проще всего познакомиться с критерием Келли в инвестиционной практике на примере игры с подбрасыванием монеты или, в более общем смысле, биномиальной модели (когда возможны только два исхода). Предположим, что вашим соперником в такой игре выступает бесконечно богатый банк или казино. Также предположим, что вероятность выпадения решки равна  $p$ , где  $\frac{1}{2} < p < 1$ . Тогда вероятность выпадения орла  $q$  составляет  $q = 1 - p < \frac{1}{2}$ . Игрок может делать ставки  $b > 0$  произвольного размера, выигрывая сумму ставки в случае удачного исхода или проигрывая все целиком в случае неудачного исхода. С учетом предполагаемых вероятностей исходов игрок, конечно же, захочет поставить на то, что выпадет решка. В результате ожидаемый выигрыш

в такой игре  $B$  (т.е. сумма случайного выигрыша в игре) при разовом подбрасывании монеты составит

$$E[B] = pb - qb = (p - q)b > 0.$$

Не будучи склонным к риску и располагая неограниченными средствами, игрок, скорее всего, захочет сделать максимально возможную ставку, чтобы получить наибольший выигрыш. Однако торговля на финансовых рынках не является разовым действием. Наоборот, это многократно повторяемые сделки. Поэтому предположим, что  $b_i$  — сумма ставки в день  $i$ , а  $c_0$  — начальный капитал. Тогда капитал  $c_1$  на конец первого дня будет определяться результатом игры только этого дня и будет равен либо  $c_0 + b_1$ , либо  $c_0 - b_1$ . Следовательно, но, ожидаемый выигрыш в игре, повторяемой  $n$  раз, составит

$$E[B^n] = c_0 + \sum_{i=1}^n (p - q)b_i.$$

В классической экономической теории предполагается, что в риск-нейтральной среде, где все трейдеры нацелены на получение прибыли, игрок будет пытаться максимизировать данное выражение. Несложно заметить, что для этого необходимо ставить все доступные средства, т.е.  $b_i = c_{i-1}$ , как и в разовой игре. Но такой алгоритм означает, что первый же проигрыш приведет к потере всех средств и банкротству игрока (если только для него не открыта безлимитная кредитная линия). Следовательно, в долгосрочной перспективе данная стратегия не позволяет максимизировать выигрыш.

Если попытка ставить весь доступный капитал неизбежно оборачивается банкротством, то нулевая ставка позволяет избежать каких бы то ни было потерь в случае проигрыша, правда, она и не позволяет ничего выиграть. Именно здесь на выручку приходит *критерий Келли*, позволяющий определить оптимальную ставку как долю  $f^*$  доступных средств в каждом раунде игры. Предположим, что  $n = h + t$ , где  $h$  — количество решек, а  $t$  — количество орлов, выпавших за  $n$  раундов игры. Тогда по истечении  $n$  раундов игры доступные средства можно рассчитать по формуле

$$c_n = c_0 (1 + f)^h (1 - f)^t.$$

В таком случае долгосрочная максимизация капитала заключается в максимизации среднего геометрического темпа роста ставки, который выражается так:

$$\begin{aligned}
r^g &= \log \left( \frac{c_n}{c_0} \right)^{1/n} = \\
&= \log \left( \frac{c_0 (1+f)^h (1-f)^t}{c_0} \right)^{1/n} = \\
&= \log \left( (1+f)^h (1-f)^t \right)^{1/n} = \\
&= \frac{h}{n} \log(1+f) + \frac{t}{n} \log(1-f).
\end{aligned}$$

Теперь задача формально сводится к максимизации *ожидаемого* среднего темпа роста за счет подбора оптимального значения параметра  $f$ . Поскольку  $E[h] = np$  и  $E[t] = nq$ , получаем

$$\begin{aligned}
E[r^g] &= E \left[ \frac{h}{n} \log(1+f) + \frac{t}{n} \log(1-f) \right] = \\
&= E[p \log(1+f) + q \log(1-f)] = \\
&= p \log(1+f) + q \log(1-f) = \\
&\equiv G(f).
\end{aligned}$$

Для максимизации полученного выражения путем выбора оптимальной доли  $f^*$  нужно проверить его первую производную, которая вычисляется так:

$$\begin{aligned}
G'(f) &= \frac{p}{1+f} - \frac{q}{1-f} = \\
&= \frac{p - pf - q - qf}{(1+f)(1-f)} = \\
&= \frac{p - q - f}{(1+f)(1-f)}.
\end{aligned}$$

Для первой производной получаем

$$G'(f) \stackrel{!}{=} 0 \Rightarrow f^* = p - q.$$

Если данное условие определяет максимум (а не минимум), то полученный результат означает, что оптимальная ставка в каждом раунде игры должна быть равна  $f^* = p - q$ . Например, при  $p = 0,55$  она будет равна  $f^* = 0,55 - 0,45 = 0,1$  (т.е. 10%).

Теперь реализуем это на Python. Как всегда, сначала импортируем пакеты и задаем конфигурационные настройки.

```
In [1]: import math
import time
import numpy as np
import pandas as pd
import datetime as dt
import cufflinks as cf
from pylab import plt
```

```
In [2]: np.random.seed(1000)
plt.style.use('seaborn')
%matplotlib inline
```

Идея заключается в том, чтобы смоделировать, например, 50 раундов по 100 подбрасываний монеты в каждом. В Python это реализуется достаточно просто.

```
In [3]: p = 0.55 ❶
```

```
In [4]: f = p - (1 - p) ❷
```

```
In [5]: f ❷
```

```
Out[5]: 0.10000000000000009
```

```
In [6]: I = 50 ❸
```

```
In [7]: n = 100 ❹
```

- ❶ Фиксированная вероятность выпадения решки.
- ❷ Расчет оптимальной ставки согласно критерию Келли.
- ❸ Количество моделируемых раундов.
- ❹ Количество подбрасываний монеты в каждом раунде.

Основной алгоритм реализован в функции `run_simulation()`, которая выполняет моделирование сделанным ранее предположениям. Результаты показаны на рис. 16.1.

```
In [8]: def run_simulation(f):
    c = np.zeros((n, I)) ❶
    c[0] = 100 ❷
    for i in range(I): ❸
```

```

        for t in range(1, n): ④
            o = np.random.binomial(1, p) ⑤
            if o > 0: ⑥
                c[t, i] = (1 + f) * c[t - 1, i] ⑦
            else: ⑧
                c[t, i] = (1 - f) * c[t - 1, i] ⑨
    return c

```

```
In [9]: c_1 = run_simulation(f) ⑩
```

```
In [10]: c_1.round(2)
```

```

Out[10]: array([[100. , 100. , 100. , ..., 100. , 100. , 100. ],
               [ 90. , 110. , 90. , ..., 110. , 90. , 110. ],
               [ 99. , 121. , 99. , ..., 121. , 81. , 121. ],
               ...,
               [226.35, 338.13, 413.27, ..., 123.97, 123.97, 123.97],
               [248.99, 371.94, 454.6 , ..., 136.37, 136.37, 136.37],
               [273.89, 409.14, 409.14, ..., 122.73, 150.01, 122.73]])

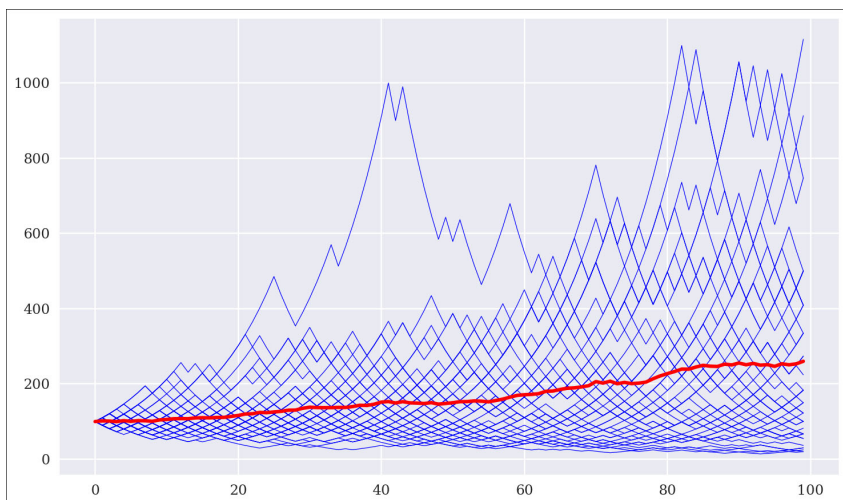
```

```

In [11]: plt.figure(figsize=(10, 6))
        plt.plot(c_1, 'b', lw=0.5) ⑪
        plt.plot(c_1.mean(axis=1), 'r', lw=2.5); ⑫

```

- ① Инициализация объекта `ndarray`, предназначенного для хранения результатов моделирования.
- ② Задание начального капитала равным 100.
- ③ Внешний цикл для раундов игры.
- ④ Внутренний цикл для подбрасываний монеты.
- ⑤ Моделирование подбрасывания монеты.
- ⑥ Если результат равен 1 (решка)...
- ⑦ ...то капитал увеличивается на сумму выигрыша (ставку).
- ⑧ Если результат равен 0 (орел)...
- ⑨ ...то проигрыш (ставка) вычитается из капитала.
- ⑩ Запуск процесса моделирования.
- ⑪ Построение графиков для 50 раундов.
- ⑫ Построение графика среднего результата за 50 раундов.



**Рис. 16.1.** Результаты моделирования 50 раундов по 100 подбрасываний монеты в каждом (линия тренда — средний результат)

В следующем коде моделирование повторяется для различных значений параметра  $f$ . Как показано на рис. 16.2, ставки, представленные меньшими долями капитала, в среднем обеспечивают более низкий темп его роста. Более высокие ставки могут привести как к увеличению среднего капитала в конце процесса моделирования ( $f = 0,25$ ), так и к существенному его уменьшению ( $f = 0,5$ ). В обоих случаях волатильность существенно повышается.

In [12]: `c_2 = run_simulation(0.05)` ❶

In [13]: `c_3 = run_simulation(0.25)` ❷

In [14]: `c_4 = run_simulation(0.5)` ❸

```
In [15]: plt.figure(figsize=(10, 6))
plt.plot(c_1.mean(axis=1), 'r', label='$f^*=0.1$')
plt.plot(c_2.mean(axis=1), 'b', label='$f=0.05$')
plt.plot(c_3.mean(axis=1), 'y', label='$f=0.25$')
plt.plot(c_4.mean(axis=1), 'm', label='$f=0.5$')
plt.legend(loc=0);
```

- ❶ Моделирование при  $f = 0,05$ .
- ❷ Моделирование при  $f = 0,25$ .
- ❸ Моделирование при  $f = 0,5$ .



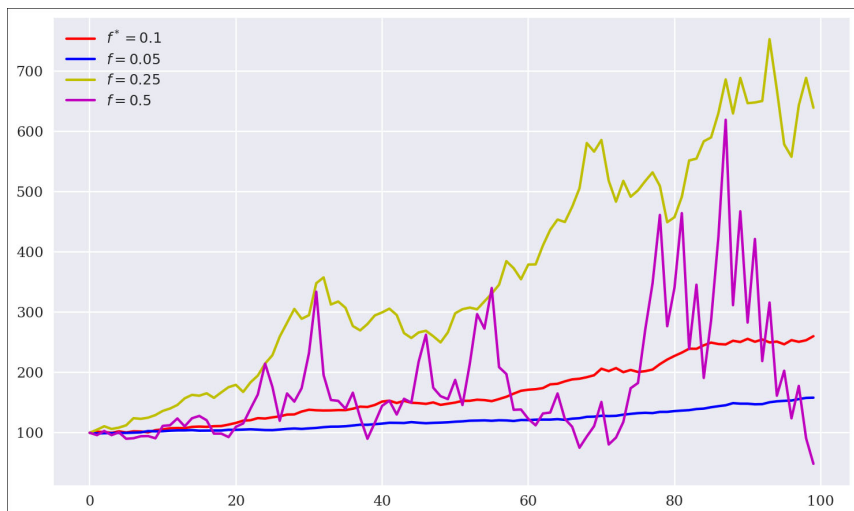


Рис. 16.2. Изменение среднего капитала при разных ставках

## Критерий Келли в биржевой торговле

Рассмотрим модель фондового рынка, в рамках которой определенная акция (индекс) с известной текущей ценой спустя год будет иметь одну из двух возможных цен. Это тоже биномиальная модель, только теперь она чуть ближе к реалиям биржевой торговли<sup>1</sup>. В частности, предположим, что

$$P(r^S = \mu + \sigma) = P(r^S = \mu - \sigma) = \frac{1}{2},$$

где  $E[r^S] = \mu > 0$  — ожидаемая доходность акций в течение одного года, а  $\sigma > 0$  — стандартное отклонение доходности (волатильность). В однопериодной модели доступный капитал по истечении года (с прежними  $c_0$  и  $f$ ) рассчитывается так:

$$c(f) = c_0(1 + (1 - f)r + fr^S),$$

где  $r$  — постоянная краткосрочная ставка для денежных средств, не инвестируемых в акции. Максимизация геометрического темпа роста означает максимизацию следующего выражения:

<sup>1</sup> Такая постановка задачи описана в работе Хунг [2].

$$G(f) = E \left[ \log \frac{c(f)}{c_0} \right].$$

Теперь предположим, что в году насчитывается  $n$  торговых дней  $i$ , для каждого из которых справедливо следующее утверждение:

$$P \left( r_i^S = \frac{\mu}{n} + \frac{\sigma}{\sqrt{n}} \right) = P \left( r_i^S = \frac{\mu}{n} - \frac{\sigma}{\sqrt{n}} \right) = \frac{1}{2}.$$

Обратите внимание на то, что волатильность возрастает пропорционально квадратному корню количества торговых дней. С учетом этого формула расчета годового капитала примет такой вид:

$$c_n(f) = c_0 \prod_{i=1}^n \left( 1 + (1-f) \frac{r}{n} + f r_i^S \right).$$

В результате для долгосрочной максимизации капитала, инвестируемого в акции, необходимо максимизировать следующую величину:

$$\begin{aligned} G_n(f) &= E \left[ \log \frac{c_n(f)}{c_0} \right] = \\ &= E \left[ \sum_{i=1}^n \log \left( 1 + (1-f) \frac{r}{n} + f r_i^S \right) \right] = \\ &= \frac{1}{2} \sum_{i=1}^n \log \left( 1 + (1-f) \frac{r}{n} + f \left( \frac{\mu}{n} + \frac{\sigma}{\sqrt{n}} \right) \right) + \log \left( 1 + (1-f) \frac{r}{n} + f \left( \frac{\mu}{n} - \frac{\sigma}{\sqrt{n}} \right) \right) = \\ &= \frac{n}{2} \log \left( \left( 1 + (1-f) \frac{r}{n} + f \frac{\mu}{n} \right)^2 - \frac{f^2 \sigma^2}{n} \right). \end{aligned}$$

Разложив это выражение в ряд Тейлора, получим:

$$G_n(f) = r + (\mu - r)f - \frac{\sigma^2}{2} f^2 + O \left( \frac{1}{\sqrt{n}} \right).$$

Для бесконечного числа торговых дней (т.е. при непрерывной торговле) оно переписывается так:

$$G_\infty(f) = r + (\mu - r)f - \frac{\sigma^2}{2} f^2.$$

Проверка первой производной такой функции показывает, что оптимальная доля  $f^*$  представляется следующим выражением:

$$f^* = \frac{\mu - r}{\sigma^2}.$$

Другими словами, это ожидаемое превышение доходности акций над безрисковой ставкой, деленное на дисперсию доходности. Данное выражение напоминает коэффициент Шарпа (см. главу 13), однако его смысл иной.

В следующем примере показано, как применять эти формулы для расчета капитала, задействуемого в торговых стратегиях. Мы рассмотрим простую торговую стратегию, которая заключается в пассивном удержании длинной позиции по индексу S&P 500. Вот как получить исходные данные и вычислить необходимые статистические показатели.

```
In [16]: raw = pd.read_csv('../source/tr_eikon_eod_data.csv',  
                           index_col=0, parse_dates=True)
```

```
In [17]: symbol = '.SPX'
```

```
In [18]: data = pd.DataFrame(raw[symbol])
```

```
In [19]: data['Доходность'] = np.log(data / data.shift(1))
```

```
In [20]: data.dropna(inplace=True)
```

```
In [21]: data.tail()
```

```
Out[21]:
```

	<b>.SPX</b>	<b>Доходность</b>
<b>Date</b>		
<b>2018-06-25</b>	2717.07	-0.013820
<b>2018-06-26</b>	2723.06	0.002202
<b>2018-06-27</b>	2699.63	-0.008642
<b>2018-06-28</b>	2716.31	0.006160
<b>2018-06-29</b>	2718.37	0.000758

Статистические показатели индекса S&P 500 за рассматриваемый период свидетельствуют о том, что оптимальная доля средств, инвестируемых в длинную позицию, составляет примерно 4,5. Другими словами, для получения дохода в 1 доллар необходимо инвестировать 4,5 доллара, что означает коэффициент леввериджа 4,5 (т.е. имеем не “долю”, а “множитель”). При прочих равных условиях критерий Келли подразумевает, что чем больше левверидж, тем выше ожидаемая доходность и тем ниже волатильность (дисперсия).

```
In [22]: mu = data['Доходность'].mean() * 252 ❶
```

```
In [23]: mu ❶  
Out[23]: 0.09898579893004976
```

```
In [24]: sigma = data['Доходность'].std() * 252 ** 0.5 ❷
```

```
In [25]: sigma ❷  
Out[25]: 0.1488567510081967
```

```
In [26]: r = 0.0 ❸
```

```
In [27]: f = (mu - r) / sigma ** 2 ❹
```

```
In [28]: f ❹  
Out[28]: 4.4672043679706865
```

- ❶ Вычисление годовой доходности.
- ❷ Вычисление годовой волатильности.
- ❸ Задание безрисковой ставки равной 0 (для простоты).
- ❹ Вычисление оптимальной доли инвестируемых средств в рамках выбранной стратегии согласно критерию Келли.

В следующем коде моделируется применение критерия Келли и расчет оптимального коэффициента леввериджа. Для простоты и удобства сравнения начальный капитал задан равным 1, а начальная инвестиция определяется как  $1 \cdot f^*$ . В зависимости от уровня доходности величина инвестиции ежедневно корректируется в соответствии с доступными средствами: в случае убытков инвестиция уменьшается, а в случае прибыли — увеличивается. Динамика изменения капитала в сравнении с самим индексом показана на рис. 16.3.

```
In [29]: equs = []
```

```
In [30]: def kelly_strategy(f):  
    global equs  
    equ = 'equity_{:.2f}'.format(f)  
    equs.append(equ)  
    cap = 'capital_{:.2f}'.format(f)  
    data[equ] = 1 ❶  
    data[cap] = data[equ] * f ❷  
    for i, t in enumerate(data.index[1:]):
```

```

t_1 = data.index[i] ❸
data.loc[t, cap] = data[cap].loc[t_1] * \
    math.exp(data['Доходность'].loc[t]) ❹
data.loc[t, equ] = data[cap].loc[t] - \
    data[cap].loc[t_1] + \
    data[equ].loc[t_1] ❺
data.loc[t, cap] = data[equ].loc[t] * f ❻

```

```
In [31]: kelly_strategy(f * 0.5) ❼
```

```
In [32]: kelly_strategy(f * 0.66) ❽
```

```
In [33]: kelly_strategy(f) ❾
```

```
In [34]: print(data[equ].tail())
           equity_2.23  equity_2.95  equity_4.47
Date
2018-06-25      4.707070      6.367340      8.794342
2018-06-26      4.730248      6.408727      8.880952
2018-06-27      4.639340      6.246147      8.539593
2018-06-28      4.703365      6.359932      8.775296
2018-06-29      4.711332      6.374152      8.805026

```

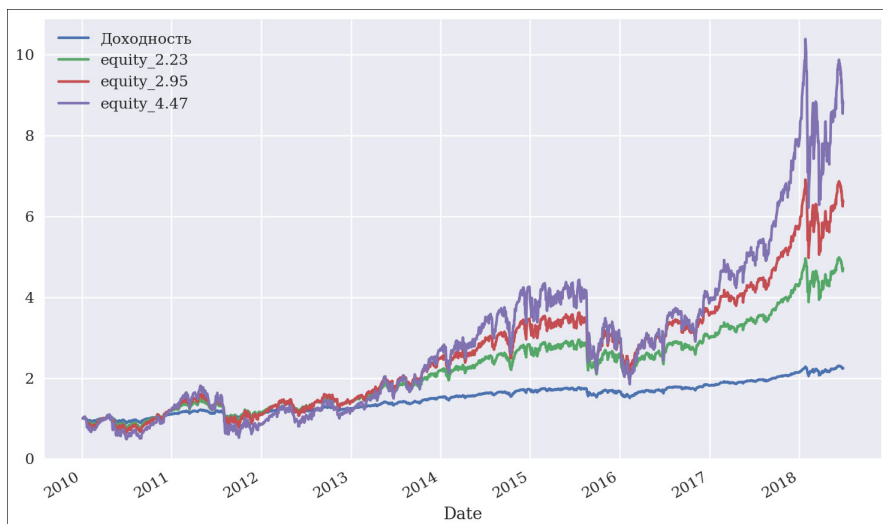
```
In [35]: ax = data['Доходность'].cumsum(
        ).apply(np.exp).plot(legend=True, figsize=(10, 6))
        data[equ].plot(ax=ax, legend=True);

```

- ❶ Генерирование нового столбца `equity` и задание начального капитала равным 1.
- ❷ Генерирование нового столбца `capital` и задание начальной инвестиции равной  $1 \cdot f^*$ .
- ❸ Подбор правильного значения `DatetimeIndex` на основе предыдущих значений.
- ❹ Расчет суммы инвестиций для полученной доходности.
- ❺ Корректировка величины капитала с учетом доходности инвестиций.
- ❻ Определение новой суммы инвестиций для скорректированной величины капитала и фиксированного коэффициента леввериджа.
- ❼ Моделирование торговой стратегии на основе критерия Келли для половины величины  $f$ ...

8 ...две трети  $f$ ...

9 ...и непосредственно  $f$ .



*Рис. 16.3. Накопительная доходность индекса S&P 500 при различных долевых инвестициях, задаваемых параметром  $f$*

Как показано на рис. 16.3, применение оптимального коэффициента леввериджа согласно критерию Келли приводит к довольно непредсказуемой динамике изменения капитала (высокой волатильности), что для коэффициента 4,47 выглядит достаточно правдоподобно. Можно ожидать, что волатильность капитала будет увеличиваться с увеличением леввериджа. Поэтому на практике инвесторы часто уменьшают коэффициент леввериджа, например до уровня “половинный Келли”, т.е.  $\frac{1}{2} f^* \approx 2,23$  в данном примере. На рис. 16.3 приведены графики для нескольких коэффициентов леввериджа, меньших  $f$ . Для них торговая стратегия оказывается менее рискованной.

## Торговая стратегия, основанная на машинном обучении

В главе 14 мы познакомились с торговой платформой FXCM, ее программным интерфейсом и оболочечным пакетом `fxstru`. В этом разделе мы дополним подход на основе машинного обучения, позволяющий прогнозировать направление движения рыночных цен, историческими данными, получаемыми

с помощью программного интерфейса FXCM, чтобы протестировать алгоритмическую торговую стратегию для валютной пары EUR/USD. Мы выполним векторизованное тестирование на исторических данных, но на этот раз будем учитывать спред как пропорциональные транзакционные издержки. Кроме того, по сравнению с простым векторизованным тестированием (см. главу 15) мы проведем более глубокий анализ рисков тестируемой стратегии.

## Векторизованное тестирование на исторических данных

Тестирование выполняется на данных торгового дня, извлекаемых с пятиминутными интервалами. В следующем коде устанавливается соединение с сервером FXCM и запрашиваются данные пятиминутных выборок за весь месяц. На рис. 16.4 приведен график среднего курса закрытия за анализируемый период.

```
In [36]: import fxcmpy
```

```
In [37]: fxcmpy.__version__
```

```
Out[37]: '1.1.33'
```

```
In [38]: api = fxcmpy.fxcmpy(config_file='../fxcm.cfg') ❶
```

```
In [39]: data = api.get_candles('EUR/USD', period='m5',
                                start='2018-06-01 00:00:00',
                                stop='2018-06-30 00:00:00') ❷
```

```
In [40]: data.iloc[-5:, 4:]
```

```
Out[40]:
```

	askopen	askclose	askhigh	asklow \
<b>date</b>				
2018-06-29 20:35:00	1.16862	1.16882	1.16896	1.16839
2018-06-29 20:40:00	1.16882	1.16853	1.16898	1.16852
2018-06-29 20:45:00	1.16853	1.16826	1.16862	1.16822
2018-06-29 20:50:00	1.16826	1.16836	1.16846	1.16819
2018-06-29 20:55:00	1.16836	1.16861	1.16876	1.16834

	tickqty
<b>date</b>	
2018-06-29 20:35:00	601
2018-06-29 20:40:00	387
2018-06-29 20:45:00	592
2018-06-29 20:50:00	842
2018-06-29 20:55:00	540

```

In [41]: data.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 6083 entries, 2018-06-01 00:00:00 to
                2018-06-29 20:55:00
Data columns (total 9 columns):
bidopen      6083 non-null float64
bidclose     6083 non-null float64
bidhigh      6083 non-null float64
bidlow       6083 non-null float64
askopen      6083 non-null float64
askclose     6083 non-null float64
askhigh      6083 non-null float64
asklow       6083 non-null float64
tickqty      6083 non-null int64
dtypes: float64(8), int64(1)
memory usage: 475.2 KB

In [42]: spread = (data['askclose'] - data['bidclose']).mean() ❷
spread ❷
Out[42]: 2.6338977478217845e-05

In [43]: data['midclose'] = (data['askclose'] + \
                             data['bidclose']) / 2 ❸

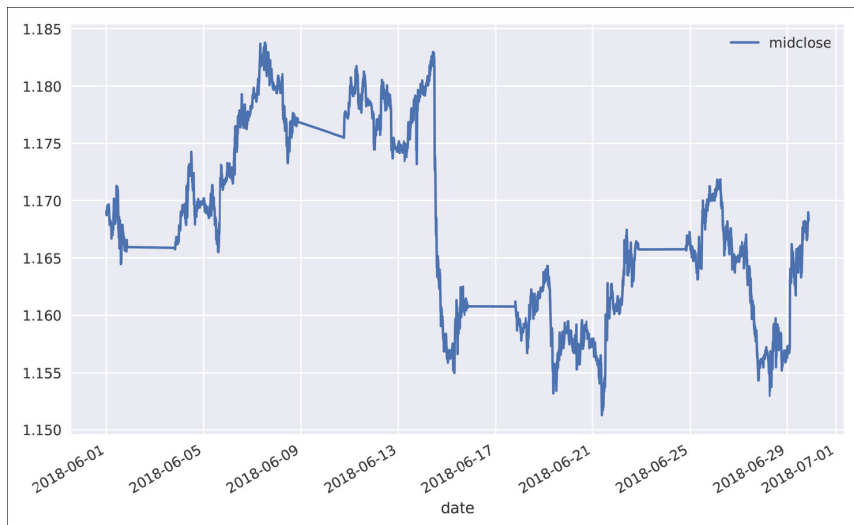
In [44]: ptc = spread / data['midclose'].mean() ❹
ptc ❹
Out[44]: 2.255685318140426e-05

In [45]: data['midclose'].plot(figsize=(10, 6), legend=True);

```

- ❶ Подключение к серверу FXCM и получение данных.
- ❷ Вычисление среднего спреда.
- ❸ Вычисление среднего курса на момент закрытия.
- ❹ Расчет средних пропорциональных транзакционных издержек на основе среднего спреда и среднего курса закрытия.





*Рис. 16.4. Валютный курс пары EUR/USD (пятиминутные выборки)*

Торговая стратегия, основанная на машинном обучении, оперирует смещенными временными рядами значений доходности, приведенных к бинарному формату. Другими словами, алгоритм машинного обучения анализирует исторические шаблоны изменения цен и на основании этого оценивает вероятность движения цены вверх или вниз. Для этого в следующем коде создаются ряды признаков, содержащие значения 0 или 1, а также ряды меток со значениями +1 или -1, указывающими направление движения.

```
In [46]: data['returns'] = np.log(data['midclose'] /
                                     data['midclose'].shift(1))
```

```
In [47]: data.dropna(inplace=True)
```

```
In [48]: lags = 5
```

```
In [49]: cols = []
          for lag in range(1, lags + 1):
              col = 'lag_{}'.format(lag)
              data[col] = data['returns'].shift(lag) ❶
              cols.append(col)
```

```
In [50]: data.dropna(inplace=True)
```

```
In [51]: data[cols] = np.where(data[cols] > 0, 1, 0) ❷
```

```
In [52]: data['direction'] = np.where(data['returns'] > 0, 1, -1) ❸
```

```
In [53]: data[cols + ['direction']].head()
```

```
Out[53]:
```

	lag_1	lag_2	lag_3	lag_4	lag_5	\
date						
2018-06-01 00:30:00	1	0	1	0	1	
2018-06-01 00:35:00	1	1	0	1	0	
2018-06-01 00:40:00	1	1	1	0	1	
2018-06-01 00:45:00	1	1	1	1	0	
2018-06-01 00:50:00	1	1	1	1	1	

	direction
date	
2018-06-01 00:30:00	1
2018-06-01 00:35:00	1
2018-06-01 00:40:00	1
2018-06-01 00:45:00	1
2018-06-01 00:50:00	-1

- ❶ Создание временного ряда значений доходности, смещенного на заданное число позиций.
- ❷ Приведение значений признака к бинарному формату.
- ❸ Преобразование значений доходности в метки направлений.

После создания рядов признаков и меток становится возможным применять различные алгоритмы машинного обучения с учителем. В следующем примере используется алгоритм классификации на основе метода опорных векторов, доступный в библиотеке Scikit-learn. Для обучения и тестирования алгоритмической торговой стратегии выполняется последовательное разделение данных на обучающий и тестовый наборы. Точность модели на обучающем наборе немного превышает 50%, тогда как на тестовых данных результат оказывается даже выше. Вместо оценок точности можно было бы вычислять *процент попаданий* торговой стратегии (т.е. количество успешных сделок в сравнении с общим числом сделок), более уместный в финансовом контексте. Поскольку в нашем случае процент попаданий больше 50%, применительно к критерию Келли это может означать небольшое преимущество стратегии по отношению к модели случайного блуждания.

```

In [54]: from sklearn.svm import SVC
         from sklearn.metrics import accuracy_score

In [55]: model = SVC(C=1, kernel='linear', gamma='auto')

In [56]: split = int(len(data) * 0.80)

In [57]: train = data.iloc[:split].copy()

In [58]: model.fit(train[cols], train['direction'])
Out[58]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovr', degree=3, gamma='auto',
            kernel='linear', max_iter=-1, probability=False,
            random_state=None, shrinking=True, tol=0.001,
            verbose=False)

In [59]: accuracy_score(train['direction'],
                        model.predict(train[cols])) ❶
Out[59]: 0.5198518823287389

In [60]: test = data.iloc[split:].copy()

In [61]: test['position'] = model.predict(test[cols])

In [62]: accuracy_score(test['direction'], test['position']) ❷
Out[62]: 0.5419407894736842

```

❶ Точность предсказаний *в пределах выборки* (обучающий набор).

❷ Точность предсказаний *вне выборки* (тестовый набор).

Хорошо известно, что процент попаданий — далеко не единственный критерий успешности торговой стратегии. Другие не менее важные показатели — уровень транзакционных издержек и правильность определения важных сделок<sup>2</sup>. Поэтому оценку торговой стратегии нужно давать путем формального векторизованного тестирования на исторических данных. В следующем коде учитываются пропорциональные транзакционные издержки, рассчитанные

---

<sup>2</sup> Неоспоримый эмпирический факт заключается в том, что для достижения высокой инвестиционной и торговой эффективности крайне важно научиться правильно определять основную направленность рынка, т.е. ключевые векторы движений как вверх, так и вниз. Этот аспект наглядно проиллюстрирован на рис. 16.5 и 16.7, где хорошо видно, что торговая стратегия неверно распознает общее направление движения базового финансового инструмента, что приводит к значительному провалу в доходности.

на основе среднего спреда. На рис. 16.5 доходность алгоритмической торговой стратегии (без учета и с учетом пропорциональных транзакционных издержек) сравнивается с доходностью пассивной эталонной инвестиции.



*Рис. 16.5. Доходность валютного курса EUR/USD и алгоритмической торговой стратегии*

```
In [63]: test['strategy'] = test['position'] * test['returns'] ❶
```

```
In [64]: sum(test['position'].diff() != 0) ❷
```

```
Out[64]: 660
```

```
In [65]: test['strategy_tc'] = np.where(test['position'].diff() != 0,
                                         test['strategy'] - ptc, ❸
                                         test['strategy'])
```

```
In [66]: test[['returns', 'strategy', 'strategy_tc']].sum(
          ).apply(np.exp)
```

```
Out[66]: returns      0.999324
strategy      1.026141
strategy_tc   1.010977
dtype: float64
```

```
In [67]: test[['returns', 'strategy', 'strategy_tc']].cumsum(
          ).apply(np.exp).plot(figsize=(10, 6));
```

- ❶ Определение значений логарифмической доходности алгоритмической торговой стратегии, основанной на машинном обучении.
- ❷ Вычисление количества сделок, определяемых торговой стратегией на основе изменений в направлениях движения цен.
- ❸ Всякий раз при совершении сделки из логарифмической доходности торговой стратегии за указанный день вычитаются пропорциональные транзакционные издержки.



### Ограничения векторизованного тестирования на исторических данных

Векторизованное тестирование на исторических данных ограничено тем, насколько близко к рыночным реалиям можно протестировать стратегию. Например, здесь невозможно напрямую учесть фиксированные транзакционные издержки по каждой сделке. В качестве аппроксимации можно учесть их косвенно, через кратное к среднему значению пропорциональных транзакционных издержек (в зависимости от среднего размера позиции). Но в целом это не самый точный подход. Если требуется более высокая точность, придется применять другие методы, например *событийное тестирование на исторических данных* с явными циклами по каждому временному интервалу.

## Оптимальный леверидж

Располагая данными о логарифмической доходности торговой стратегии, можно определить среднее и дисперсию, чтобы вычислить оптимальный леверидж в соответствии с критерием Келли. В следующем коде данные масштабируются до годовых значений, что, впрочем, не влияет на расчет оптимального левериджа, поскольку средняя доходность и дисперсия масштабируются одинаковыми множителями.

```
In [68]: mean = test[['returns', 'strategy_tc']].mean() *
          len(data) * 12 ❶
          mean
Out[68]: returns      -0.040535
          strategy_tc   0.654711
          dtype: float64

In [69]: var = test[['returns', 'strategy_tc']].var() * \
          len(data) * 12 ❷
          var
```

```
Out[69]: returns      0.007861
         strategy_tc  0.007837
         dtype: float64
```

```
In [70]: vol = var ** 0.5 ❸
         vol
```

```
Out[70]: returns      0.088663
         strategy_tc  0.088524
         dtype: float64
```

```
In [71]: mean / var ❹
```

```
Out[71]: returns      -5.156448
         strategy_tc  83.545792
         dtype: float64
```

```
In [72]: mean / var * 0.5 ❺
```

```
Out[72]: returns      -2.578224
         strategy_tc  41.772896
         dtype: float64
```

- ❶ Среднегодовые доходности.
- ❷ Годовые дисперсии.
- ❸ Годовые волатильности.
- ❹ Оптимальный левеидж согласно критерию Келли (“полный Келли”).
- ❺ Оптимальный левеидж согласно критерию Келли (“половинный Келли”).

В случае “половинного Келли” оптимальный левеидж для нашей торговой стратегии примерно равен 40. Благодаря торговым площадкам типа FXCM и таким финансовым инструментам, как форекс и контракты на разницу цен (Contract For Difference — CFD), это вполне достижимый уровень даже для розничных трейдеров<sup>3</sup>. На рис. 16.6 сравнивается доходность торговой стратегии, учитывающей транзакционные издержки, для разных значений левеиджа.

---

<sup>3</sup> Увеличение левеиджа приводит к существенному повышению рисков. Трейдеры должны внимательно читать все предупреждения, касающиеся рисков и регуляторной политики. Положительные результаты ретроспективного тестирования не гарантируют получения дохода в будущем. Все приведенные здесь результаты призваны всего лишь продемонстрировать возможности рассматриваемых программных методик и аналитических подходов. В некоторых странах, например в Германии, вводятся ограничения на допустимый уровень левеиджа для розничных трейдеров, в зависимости от категории финансовых инструментов.

```
In [73]: to_plot = ['returns', 'strategy_tc']
```

```
In [74]: for lev in [10, 20, 30, 40, 50]:  
    label = 'lstrategy_tc_%d' % lev  
    test[label] = test['strategy_tc'] * lev ❶  
    to_plot.append(label)
```

```
In [75]: test[to_plot].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Масштабирование доходностей стратегии для разных значений левериджа.

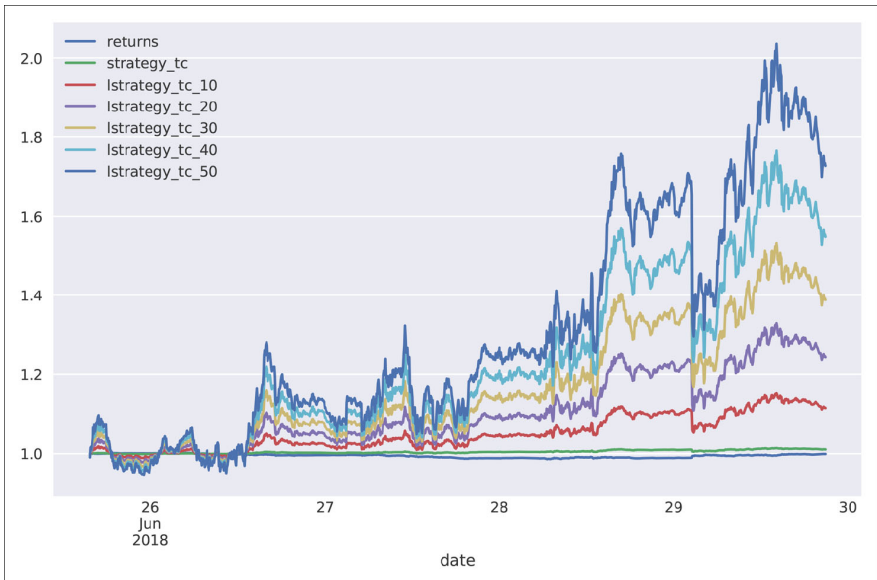


Рис. 16.6. Доходность алгоритмической торговой стратегии при разных значениях левериджа

## Анализ рисков

Поскольку увеличение левериджа приводит к повышению рисков, связанных с торговой стратегией, имеет смысл выполнить углубленный анализ рисков. В следующем примере предполагается, что коэффициент левериджа равен 30. Для начала вычислим максимальную просадку и самый длительный период просадки. *Максимальная просадка* (дродаун) — это наибольшее снижение капитала после недавнего подъема. Соответственно, *самый длительный период просадки* — это наибольший интервал времени, требуемый для

возврата к подъему. В данном примере стартовый баланс равен 3333 евро, что приводит к начальной позиции в 100 тыс. евро при коэффициенте леввериджа 30. Предполагается, что стартовый баланс остается неизменным вне зависимости от доходности.

```
In [76]: equity = 3333 ❶
```

```
In [77]: risk = pd.DataFrame(test['lstrategy_tc_30']) ❷
```

```
In [78]: risk['equity'] = risk['lstrategy_tc_30'].cumsum(  
        ).apply(np.exp) * equity ❸
```

```
In [79]: risk['cummax'] = risk['equity'].cummax() ❹
```

```
In [80]: risk['drawdown'] = risk['cummax'] - risk['equity'] ❺
```

```
In [81]: risk['drawdown'].max() ❻  
Out[81]: 781.7073602069818
```

```
In [82]: t_max = risk['drawdown'].idxmax() ❼  
        t_max ❼  
Out[82]: Timestamp('2018-06-29 02:45:00')
```

- ❶ Начальный капитал.
- ❷ Временной ряд со значениями логарифмической доходности...
- ❸ ...масштабируемыми на величину начального капитала.
- ❹ Накопительные максимальные значения.
- ❺ Просадки.
- ❻ Максимальная просадка.
- ❼ Момент времени, когда это случилось.

С технической точки зрения (очередной) пик характеризуется нулевой просадкой. Период просадки — это интервал времени между двумя такими пиками. На рис. 16.7 визуализируются периоды просадки и показана точка максимальной просадки.



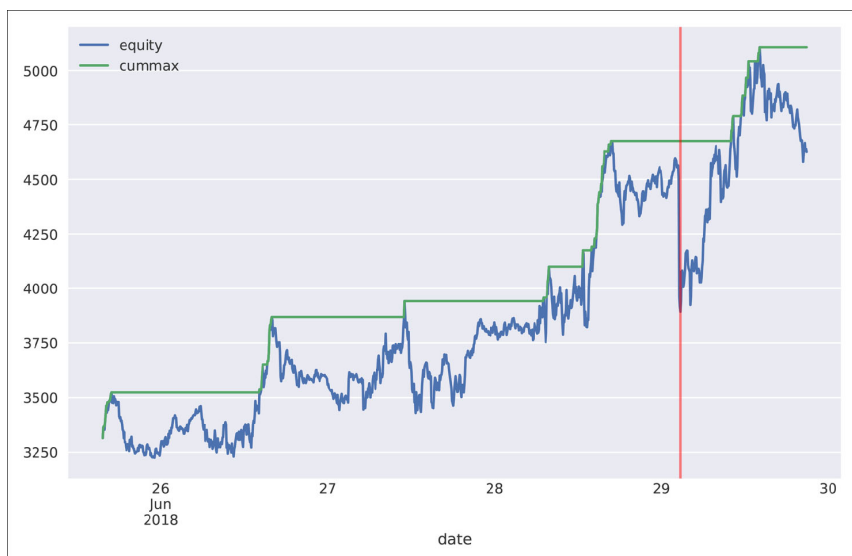


Рис. 16.7. Максимальная просадка (вертикальная линия) и периоды просадки (горизонтальные линии)

```
In [83]: temp = risk['drawdown'][risk['drawdown'] == 0] ❶
```

```
In [84]: periods = (temp.index[1:].to_pydatetime() -
                    temp.index[:-1].to_pydatetime()) ❷
```

```
In [85]: periods[20:30] ❷
```

```
Out[85]: array([datetime.timedelta(seconds=68700),
                datetime.timedelta(seconds=72000),
                datetime.timedelta(seconds=1800),
                datetime.timedelta(seconds=300),
                datetime.timedelta(seconds=600),
                datetime.timedelta(seconds=300),
                datetime.timedelta(seconds=17400),
                datetime.timedelta(seconds=4500),
                datetime.timedelta(seconds=1500),
                datetime.timedelta(seconds=900)], dtype=object)
```

```
In [86]: t_per = periods.max() ❸
```

```
In [87]: t_per ❸
```

```
Out[87]: datetime.timedelta(seconds=76500)
```

```
In [88]: t_per.seconds / 60 / 60 ④
```

```
Out[88]: 21.25
```

```
In [89]: risk[['equity', 'cummax']].plot(figsize=(10, 6))  
        plt.axvline(t_max, c='r', alpha=0.5);
```

- ① Выявление пиков с нулевой просадкой.
- ② Вычисление значений `timedelta` между всеми пиками.
- ③ Самый длительный период просадки в секундах...
- ④ ...и часах.

Еще одна важная мера риска — это *стоимость под риском* (Value at Risk — VaR). Она определяется как максимальная величина потерь, ожидаемых за определенный период времени с заданной вероятностью. В следующем примере значения VaR рассчитываются на основе данных о логарифмической доходности торговой стратегии при различных доверительных вероятностях (рассматривается фиксированный период времени, соответствующий пяти-минутной выборке).

```
In [90]: import scipy.stats as scs
```

```
In [91]: percs = np.array([0.01, 0.1, 1., 2.5, 5.0, 10.0]) ①
```

```
In [92]: risk['returns'] = np.log(risk['equity'] /  
                                risk['equity'].shift(1))
```

```
In [93]: VaR = scs.scoreatpercentile(equity * risk['returns'],  
                                     percs) ②
```

```
In [94]: def print_var():  
    print('%16s %16s' % ('Confidence Level',  
                        'Value-at-Risk'))  
    print(33 * '-')  
    for pair in zip(percs, VaR):  
        print('%16.2f %16.3f' % (100 - pair[0], -pair[1])) ③
```

```
In [95]: print_var() ③
```

Confidence Level	Value-at-Risk
-----	-----
99.99	400.854
99.90	175.932
99.00	88.139
97.50	60.485
95.00	45.010
90.00	32.056

- ❶ Используемые процентиля.
- ❷ Вычисление значений VaR для заданных процентилей.
- ❸ Перевод процентилей в доверительные вероятности и (отрицательных) значений VaR в положительные.

Наконец, в следующем коде значения VaR вычисляются для временного горизонта длительностью один час (выполняется перевыборка исходного объекта `DataFrame`). В результате стоимость под риском увеличивается для всех доверительных вероятностей, кроме самой высокой.

```
In [96]: hourly = risk.resample('1H', label='right').last() ❶
```

```
In [97]: hourly['returns'] = np.log(hourly['equity'] /
                                     hourly['equity'].shift(1))
```

```
In [98]: VaR = scs.scoreatpercentile(equity * hourly['returns'],
                                     percs) ❷
```

```
In [99]: print_var()
Confidence Level  Value-at-Risk
-----
```

99.99	389.524
99.90	372.657
99.00	205.662
97.50	186.999
95.00	164.869
90.00	101.835

- ❶ Повторная выборка данных с часовыми интервалами вместо пятиминутных.
- ❷ Пересчет значений VaR для обновленных данных.

## Сохранение объекта модели

После того как алгоритмическая торговая стратегия “принимается” по результатам тестирования на исторических данных, расчета оптимального леве-риджа и анализа рисков, объект модели можно сохранить для последующего использования в производственной среде.

```
In [100]: import pickle
```

```
In [101]: pickle.dump(model, open('algorithm.pkl', 'wb'))
```

## Веб-алгоритм

До этого момента мы тестировали алгоритм торговой стратегии в *автономном режиме*. Другими словами, для решения поставленной задачи ему требовался полный набор данных. Основная трудность заключалась в обучении SVM-алгоритма на бинарных признаках и метках, задающих направление движения. На практике при развертывании системы алгоритмической торговли для работы на финансовых рынках данные будут обрабатываться по мере поступления, и предсказываться будет направление рыночного движения в следующем временном интервале. В этом разделе мы интегрируем сохраненный объект модели в среду потоковой обработки данных.

Программа, которая преобразует автономный алгоритм в систему реального времени, должна решить следующие основные задачи.

### *Обработка тиковых данных*

Тиковые данные поступают в реальном времени и должны обрабатываться в потоковом режиме.

### *Повторная выборка данных*

Тиковые данные нужно перевыбирать в соответствии с величиной интервала, установленного в алгоритме торговой стратегии.

### *Прогнозирование*

Торговый алгоритм прогнозирует направление рыночного движения в будущие периоды.

### *Размещение заявок*

С учетом текущей позиции и сгенерированного прогноза (“сигнала”) программа должна либо размещать заявку, либо удерживать позицию.

В главе 14 было показано, как получать тиковые данные с помощью программного интерфейса FXCM в режиме реального времени. Идея заключалась в том, чтобы подписаться на поток рыночных данных и зарегистрировать функцию обратного вызова, которая будет обрабатывать эти данные.

В первую очередь необходимо загрузить сохраненный объект алгоритмической модели, в котором реализована вся необходимая торговая логика. Будет также полезно создать вспомогательную функцию, выводящую сведения об открываемых позициях в момент подачи алгоритмом соответствующих заявок.

```
In [102]: algorithm = pickle.load(open('algorithm.pkl', 'rb'))
```

```
In [103]: algorithm
```

```
Out[103]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,  
             decision_function_shape='ovr', degree=3, gamma='auto',  
             kernel='linear', max_iter=-1, probability=False,  
             random_state=None, shrinking=True, tol=0.001,  
             verbose=False)
```

```
In [104]: sel = ['tradeId', 'amountK', 'currency',  
                'grossPL', 'isBuy'] ❶
```

```
In [105]: def print_positions(pos):  
            print('\n\n' + 50 * '=')  
            print('Going {}'.format(pos))  
            time.sleep(1.5) ❷  
            print(api.get_open_positions()[sel]) ❸  
            print(50 * '=' + '\n\n')
```

- ❶ Выводимые на экран столбцы объекта DataFrame.
- ❷ Ожидание выполнения заявки и обновления открытых позиций.
- ❸ Вывод информации об открытых позициях.

Прежде чем запускать веб-алгоритм, необходимо задать ряд параметров.

```
In [106]: symbol = 'EUR/USD' ❶  
          bar = '15s' ❷  
          amount = 100 ❸  
          position = 0 ❹  
          min_bars = lags + 1 ❺  
          df = pd.DataFrame() ❻
```

- ❶ Тикер торгуемого финансового инструмента.
- ❷ Интервал повторной выборки данных. Для простоты тестирования этот интервал должен быть короче реального периода (например, 15 секунд вместо 5 минут).
- ❸ Величина денежных средств, в тысячах.
- ❹ Начальная позиция (нейтральная).
- ❺ Минимальное количество интервалов повторной выборки, обеспечивающих начальный прогноз и запуск торгов.
- ❻ Пустой объект DataFrame, впоследствии заполняемый повторно выбираемыми данными.

Ниже приведен код функции обратного вызова `automated_strategy()`, которая позволяет применять торговый алгоритм в реальном времени.

```
In [107]: def automated_strategy(data, dataframe):
    global min_bars, position, df
    ldf = len(dataframe) ❶
    df = dataframe.resample(bar,
                            label='right').last().ffill() ❷

    if ldf % 20 == 0:
        print('%3d' % len(dataframe), end=',')

    if len(df) > min_bars:
        min_bars = len(df)
        df['Mid'] = df[['Bid', 'Ask']].mean(axis=1)
        df['Returns'] = np.log(df['Mid'] /
                               df['Mid'].shift(1))
        df['Direction'] = np.where(df['Returns'] > 0, 1, -1)
        features = df['Direction'].iloc[-(lags + 1):-1] ❸
        features = features.values.reshape(1, -1) ❹
        signal = algorithm.predict(features)[0] ❺

        if position in [0, -1] and signal == 1: ❻
            api.create_market_buy_order(
                symbol, amount - position * amount)
            position = 1
            print_positions('LONG')

        elif position in [0, 1] and signal == -1: ❼
            api.create_market_sell_order(
                symbol, amount + position * amount)
```

```

position = -1
print_positions('SHORT')

if len(dataframe) > 350: ❸
    api.unsubscribe_market_data('EUR/USD')
    api.close_all()

```

- ❶ Определение длины объекта DataFrame, содержащего тиковые данные.
- ❷ Повторная выборка тиковых данных в соответствии с заданным интервалом.
- ❸ Выбор подходящих значений признаков для всех смещенных выборок...
- ❹ ...и изменение формы объекта, чтобы его можно было применять для прогнозирования.
- ❺ Генерирование прогноза (+1 или -1).
- ❻ Условие открытия (или удержания) *длинной* позиции.
- ❼ Условие открытия (или удержания) *короткой* позиции.
- ❽ Условие прекращения торгов и закрытия всех открытых позиций (определяется произвольным образом по числу полученных тиков).

## Инфраструктура и развертывание

Для развертывания автоматизированной стратегии алгоритмической торговли с подключением реальных денежных средств требуется соответствующая инфраструктура, которая, помимо всего прочего, должна удовлетворять следующим условиям.

### Надежность

Инфраструктура, предназначенная для развертывания алгоритмической торговой стратегии, должна обеспечивать высокую доступность (например, > 99,9%) и гарантировать надежное хранение данных (автоматическое резервное копирование, избыточность дисков и веб-подключений и т.п.).

### Производительность

В зависимости от объемов обрабатываемых данных и выполняемых расчетов инфраструктура должна обладать достаточной вычислительной мощностью (количество ядер CPU, объем оперативной памяти, емкость

хранилища SSD). Кроме того, веб-каналы должны быть достаточно скоростными.

### Безопасность

Операционная система и запущенные на ней приложения должны быть защищены надежными паролями, а также поддерживать SSL-шифрование. Оборудование должно быть защищено от огня, воды и несанкционированного физического доступа.

По большому счету, все эти требования можно соблюсти только путем аренды соответствующей инфраструктуры в профессиональном дата-центре или облачном хранилище. Инвестиции в физическую инфраструктуру такого уровня будут оправданы только для крупнейших игроков финансового рынка.

Для задач тестирования подойдет даже самый маленький дроплет (облачный экземпляр) на сайте DigitalOcean. На момент написания книги аренда такого дроплета стоила всего 5 долларов в месяц. При этом тарификация выполняется на почасовой основе. Сервер на дроплете создается за считанные минуты, а удаляется — всего за пару секунд<sup>4</sup>.

Подробнее о том, как создать дроплет на сайте DigitalOcean, рассказывалось в главе 2. Там были приведены соответствующие *bash*-сценарии, которые можно адаптировать под индивидуальные требования (в частности, указать подключаемые пакеты Python).



### Операционные риски

Автоматизированные стратегии алгоритмической торговли можно разрабатывать и тестировать на локальном компьютере (настольный ПК, ноутбук и т.п.), но когда дело касается развертывания онлайн-систем, оперирующих реальными денежными средствами, ни о каких локальных системах не может идти речи. Перебои с подключением к Интернету или кратковременные сбои питания могут полностью нарушить работу системы, вызвав появление непреднамеренно открытых позиций или повреждение набора обрабатываемых данных (из-за пропадания тиковых данных, поступающих в реальном времени). В результате система может начать генерировать неправильные сигналы и совершать нежелательные сделки.

---

<sup>4</sup> Новые пользователи, которые регистрируются по ссылке [https://bit.ly/do\\_sign\\_up](https://bit.ly/do_sign_up), получают 60-дневный приветственный бонус в размере 100 долларов.



## Протоколирование и мониторинг

Предположим, что автоматизированная стратегия алгоритмической торговли развертывается на удаленном сервере (облачный экземпляр, арендованный сервер и т.п.), на котором уже установлены все требуемые пакеты Python (см. главу 2) и выполняется защищенная рабочая среда Jupyter Notebook (<http://bit.ly/2A8jkDx>). Что еще должен сделать алгоритмический трейдер, чтобы не сидеть круглые сутки перед монитором?

В этом разделе мы рассмотрим две важные задачи: *протоколирование и мониторинг операций в реальном времени*. Протоколирование позволяет сохранить сведения обо всех важных событиях на диске для дальнейшего изучения. Это стандартная процедура при разработке и развертывании любого программного обеспечения. Но в нашем случае протоколирование будет больше нацелено на регистрацию важных финансовых данных и событий. То же самое касается мониторинга сетевой активности в реальном времени. Можно создать сетевые каналы для непрерывного контроля важных финансовых параметров на локальном компьютере, даже когда система развернута в облаке.

В конце главы будет приведен сценарий Python, в котором реализуется все вышесказанное и используется код рассмотренного ранее веб-алгоритма. Сценарий обеспечивает развертывание автоматизированной торговой стратегии (на основе ранее сохраненного объекта алгоритмической модели) на удаленном сервере. Он также снабжает трейдера инструментами протоколирования и мониторинга благодаря пользовательской функции, которая, помимо всего прочего, использует библиотеку ZeroMQ (<https://zeromq.org/>) для создания сокетов. В сочетании со вторым, более коротким сценарием из раздела “Мониторинг стратегии” это позволяет выполнять дистанционный мониторинг всех событий, возникающих на удаленном сервере, в режиме реального времени.

При запуске сценария (либо локально, либо удаленно) по сети будет передан журнал примерно такого вида.

```
2018-07-25 09:16:15.568208
```

```
=====
NUMBER OF BARS: 24
```

```
=====
MOST RECENT DATA
```

	Mid	Returns	Direction
2018-07-25 07:15:30	1.168885	-0.000009	-1
2018-07-25 07:15:45	1.168945	0.000043	1
2018-07-25 07:16:00	1.168895	-0.000051	-1

```
2018-07-25 07:16:15 1.168895 -0.000009 -1
2018-07-25 07:16:30 1.168885 -0.000017 -1
```

```
=====
features: [[ 1 -1 1 -1 -1]]
position: -1
signal: -1
```

```
2018-07-25 09:16:15.581453
```

```
=====
no trade placed
```

```
****END OF CYCLE****
```

```
2018-07-25 09:16:30.069737
```

```
=====
NUMBER OF BARS: 25
```

```
=====
MOST RECENT DATA
```

	Mid	Returns	Direction
2018-07-25 07:15:45	1.168945	0.000043	1
2018-07-25 07:16:00	1.168895	-0.000051	-1
2018-07-25 07:16:15	1.168895	-0.000009	-1
2018-07-25 07:16:30	1.168950	0.000034	1
2018-07-25 07:16:45	1.168945	-0.000017	-1

```
=====
features: [[-1 1 -1 -1 1]]
position: -1
signal: 1
```

```
2018-07-25 09:16:33.035094
```

```
=====
Going LONG.
```

tradeId	amountK	currency	grossPL	isBuy
0 61476318	100	EUR/USD	-2	True

```
=====
****END OF CYCLE****
```

Последующий запуск сценария из раздела “Мониторинг стратегии” на локальном компьютере позволит получать и обрабатывать такую информацию в реальном времени. Разумеется, протоколирование можно легко настроить под конкретные задачи<sup>5</sup>. Можно также дополнительно сохранять объекты DataFrame, создаваемые в ходе выполнения торгового сценария. Более того, всю логику алгоритмической торговли можно функционально расширить, добавив такие возможности, как стоп-лоссы и программное достижение заданных уровней прибыли. Или же можно задействовать более сложные типы заявок, доступные через программный интерфейс FXCM (<http://fxcmpy.tpq.io/>).



### Учет всех рисков

Торговля валютными парами и/или контрактами на разницу цен (CFD) связана с целым рядом финансовых рисков. Реализация автоматической стратегии алгоритмической торговли для таких инструментов ведет к дополнительным рискам, среди которых самые опасные — ошибки логики торгов или исполнения заявок. Также следует учитывать технические риски, например проблемы с сетевыми подключениями и задержки с получением данных. Следовательно, перед развертыванием автоматизированной торговой стратегии нужно убедиться в том, что все рыночные, операционные, технические и прочие риски выявлены, оценены и устранены. Рассмотренный в этой главе программный код призван продемонстрировать только технические возможности алгоритмических торговых стратегий.

## Резюме

Эта глава была посвящена автоматизированному развертыванию алгоритмической торговой стратегии, основанной на машинном обучении. Применяемый алгоритм классификации позволяет прогнозировать направление изменения рыночных цен. В главе рассматривались такие важные темы, как управление капиталом (на основе критерия Келли), векторизованное тестирование на исторических данных, анализ рисков, переход от автономной к потоковой обработке финансовых данных, развертывание инфраструктуры алгоритмической торговли и протоколирование выполняемых при этом операций.

---

<sup>5</sup> Обратите внимание на то, что в обоих сценариях сетевые подключения не шифруются, из-за чего по сети передается простой текст. В производственных условиях это создает угрозу безопасности.

Все затронутые в главе темы достаточно сложные и требуют практического опыта алгоритмической торговли. С другой стороны, доступность веб-интерфейсов алгоритмической торговли, подобных тем, которые предлагаются платформой FXCM, значительно упрощает задачу автоматизации, поскольку все в основном сводится к использованию оболочечного пакета `fxcmru` для получения тиковых данных и размещения заявок. К этому программному ядру остается добавить средства максимального смягчения операционных и технических рисков.

## Сценарии Python

### Автоматизированная торговая стратегия

Ниже приведен сценарий Python, предназначенный для реализации автоматизированной стратегии алгоритмической торговли, включая функции протоколирования и мониторинга.

```
#
# Автоматизированная стратегия алгоритмической
# торговли для FXCM на основе машинного обучения.
# Веб-алгоритм, протоколирование и мониторинг.
#

# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import zmq
import time
import pickle
import fxcmru
import numpy as np
import pandas as pd
import datetime as dt

sel = ['tradeId', 'amountK', 'currency', 'grossPL', 'isBuy']

log_file = 'automated_strategy.log'

# Загрузка сохраненного объекта модели
algorithm = pickle.load(open('algorithm.pkl', 'rb'))
```

```

# Настройка сетевого подключения через библиотеку ZeroMQ
# (здесь: "издатель")
context = zmq.Context()
socket = context.socket(zmq.PUB)

# Привязка сокета ко всем IP-адресам устройства
socket.bind('tcp://0.0.0.0:5555')

def logger_monitor(message, time=True, sep=True):
    ''' Пользовательская функция протоколирования и мониторинга.
    ...

    with open(log_file, 'a') as f:
        t = str(dt.datetime.now())
        msg = ''
        if time:
            msg += '\n' + t + '\n'
        if sep:
            msg += 66 * '=' + '\n'
        msg += message + '\n\n'
        # Отправка сообщения через сокет
        socket.send_string(msg)
        # Запись сообщения в файл журнала
        f.write(msg)

def report_positions(pos):
    ''' Вывод, протоколирование и отправка данных о позициях
    ...

    out = '\n\n' + 50 * '=' + '\n'
    out += 'Going {}.{}\n'.format(pos) + '\n'
    time.sleep(2) # ожидание исполнения заявки
    out += str(api.get_open_positions()[sel]) + '\n'
    out += 50 * '=' + '\n'
    logger_monitor(out)
    print(out)

def automated_strategy(data, dataframe):
    ''' Функция обратного вызова, реализующая торговую логику
    ...

    global min_bars, position, df
    # Повторная выборка тиковых данных
    df = dataframe.resample(bar, label='right').last().ffill()

    if len(df) > min_bars:
        min_bars = len(df)

```

```

logger_monitor('NUMBER OF TICKS: {} | '.format(len(dataframe)) +
               'NUMBER OF BARS: {}'.format(min_bars))
# Обработка данных и подготовка признаков
df['Mid'] = df[['Bid', 'Ask']].mean(axis=1)
df['Returns'] = np.log(df['Mid'] / df['Mid'].shift(1))
df['Direction'] = np.where(df['Returns'] > 0, 1, -1)
# Выбор подходящих точек
features = df['Direction'].iloc[-(lags + 1):-1]
# Необходимое изменение формы набора
features = features.values.reshape(1, -1)
# Генерирование сигнала (+1 или -1)
signal = algorithm.predict(features)[0]

# Протоколирование и отправка основных финансовых данных
logger_monitor('MOST RECENT DATA\n' +
               str(df[['Mid', 'Returns',
                       'Direction']].tail()), False)
logger_monitor('features: ' + str(features) + '\n' +
               'position: ' + str(position) + '\n' +
               'signal: ' + str(signal), False)

# Торговая логика
if position in [0, -1] and signal == 1: # длинная позиция?
    api.create_market_buy_order(
        symbol, size - position * size) # размещение заявки
                                         # на покупку
    position = 1 # изменение позиции на длинную
    report_positions('LONG')

elif position in [0, 1] and signal == -1: # короткая позиция?
    api.create_market_sell_order(
        symbol, size + position * size) # размещение заявки
                                         # на продажу
    position = -1 # изменение позиции на короткую
    report_positions('SHORT')
else: # заявка не требуется
    logger_monitor('no trade placed')

logger_monitor('****END OF CYCLE***\n\n', False, False)

if len(dataframe) > 350: # условие прекращения торгов
    api.unsubscribe_market_data('EUR/USD') # отказ от получения
                                           # потоковых данных

```

```

report_positions('CLOSE OUT')
api.close_all() # закрытие всех открытых позиций
logger_monitor('***CLOSING OUT ALL POSITIONS***')

if __name__ == '__main__':
    symbol = 'EUR/USD' # тикер торгуемого финансового инструмента
    bar = '15s'        # интервал выборки; скорректируйте
                        # для тестирования и развертывания
    size = 100         # размер позиции в тысячах денежных единиц
    position = 0       # начальная позиция
    lags = 5           # количество смещенных позиций
                        # для генерирования признаков
    min_bars = lags + 1 # минимальная длина объекта DataFrame,
                        # содержащего повторно выбранные данные
    df = pd.DataFrame()
    # Задайте путь к конфигурационному файлу
    api = fxcm.py.fxcm.py(config_file='../fxcm.cfg')
    # Основной асинхронный цикл с функцией обратного вызова
    api.subscribe_market_data(symbol, (automated_strategy,))

```

## Мониторинг стратегии

Ниже приведен сценарий Python, предназначенный для реализации локального или удаленного мониторинга автоматизированной стратегии алгоритмической торговли через сетевое подключение.

```

#
# Автоматизированная стратегия алгоритмической
# торговли для FXCM на основе машинного обучения.
# Мониторинг стратегии через сетевое подключение.
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import zmq

# Настройка сетевого подключения через библиотеку ZeroMQ
# (здесь: "подписчик")
context = zmq.Context()
socket = context.socket(zmq.SUB)

# Задайте IP-адрес удаленного сервера
socket.connect('tcp://REMOTE_IP_ADDRESS:5555')

```

```
# Настройка сокета на получение любых сообщений
socket.setsockopt_string(zmq.SUBSCRIBE, '')

while True:
    msg = socket.recv_string()
    print(msg)
```

## Дополнительные ресурсы

В главе упоминались следующие источники.

1. Rotando, Louis, and Edward Thorp. *The Kelly Criterion and the Stock Market* (1992, *The American Mathematical Monthly*, Vol. 99, No. 10, pp. 922–931).
2. Hung, Jane. *Betting with the Kelly Criterion* (2010, [http://bit.ly/betting\\_with\\_kelly](http://bit.ly/betting_with_kelly)).

Сертификационный онлайн-курс, посвященный алгоритмической торговле на Python, доступен на сайте <http://certificate.tpq.io>.





# Анализ деривативов

Эта часть книги посвящена разработке небольшого, но достаточно эффективного приложения для оценки стоимости опционов и других деривативов по методу Монте-Карло<sup>1</sup>. Нашей целью будет создание библиотеки DX (Derivatives analytiX), обладающей следующими возможностями.

## *Построение модели*

Создание модели расчета краткосрочных ставок для задач дисконтирования. Моделирование европейских и американских опционов с учетом рисков и других рыночных факторов. Моделирование сложных портфелей опционов, подверженных сразу нескольким (возможно, взаимосвязанным) факторам риска.

## *Моделирование рисков*

Моделирование факторов риска на основе геометрического броуновского движения, прыжковой диффузии и диффузии по закону квадратного корня. Учет одновременного влияния сразу нескольких факторов риска независимо от наличия корреляции между ними.

## *Оценка стоимости опционов*

Риск-нейтральная оценка европейских и американских опционов с произвольными выплатами. Общая оценка портфелей, состоящих из таких опционов.

## *Управление рисками*

Численная оценка наиболее важных греческих коэффициентов — дельты и веги — независимо от основного фактора риска или типа опциона.

---

<sup>1</sup> В качестве введения в торговлю опционами и смежные темы, такие как основы функционирования рынков и роль греческих коэффициентов в управлении опционами, можно порекомендовать книгу Биттмана [1].

## *Практические задачи*

Использование специального пакета для управления портфелем внебиржевых американских опционов на фондовый индекс DAX по рыночным принципам — на основе откалиброванной модели индекса DAX.

Мы воспользуемся пакетом DX Analytics (<http://dx-analytics.com/>), который разработан и поддерживается автором книги и компанией The Python Quants GmbH (доступен через авторскую платформу Quant; <http://pqr.io/>). Его полнофункциональная версия позволяет, в частности, моделировать сложные мультирисковые деривативы и портфели таких деривативов.

Данная часть включает следующие главы.

- **Глава 17** содержит описание теоретических и технических принципов оценки опционов. Теоретическая часть представлена фундаментальной теоремой ценообразования финансовых активов и риск-нейтральным подходом к оценке стоимости опционов. В технической части мы рассмотрим классы Python, предназначенные для риск-нейтрального дисконтирования и учета рыночных факторов.
- **Глава 18** посвящена моделированию факторов риска на основе геометрического броуновского движения, прыжковой диффузии и диффузии по закону квадратного корня. Для этого будут созданы один общий и три специализированных класса Python.
- **Глава 19** посвящена оценке европейских и американских опционов с одним основным фактором риска. Для этого будут созданы один общий и два специализированных класса Python. Общий класс позволяет оценить коэффициенты дельта и вега независимо от типа опциона.
- **Глава 20** посвящена методам оценки сложных портфелей деривативов с множеством потенциально коррелированных факторов риска. Будет создан простой класс Python, предназначенный для моделирования поведения дериватива, а также более сложный класс, предназначенный для согласованной оценки всего портфеля.
- **Глава 21** посвящена использованию библиотеки DX, разработанной в предыдущих главах, для оценки портфеля американских пут-опционов на основе фондового индекса DAX.

---

# Принципы оценки опционов

Сложные проценты — величайшее математическое открытие всех времен.

*Альберт Эйнштейн*

В этой главе мы приступим к разработке библиотеки DX, изучив базовые концепции анализа деривативов. Сначала мы вкратце рассмотрим фундаментальную теорему ценообразования финансовых активов, которая служит теоретической основой для моделирования и оценки опционов. Затем мы поговорим о решении таких ключевых задач, как *обработка дат* и *риск-нейтральное дисконтирование*. Мы будем учитывать только простейший случай — дисконтирование с постоянной краткосрочной ставкой. Включить в библиотеку более сложные и реалистичные модели не составит особого труда. Также мы рассмотрим концепцию *рыночной среды*, т.е. совокупности констант, списков и кривых, которые понадобятся для создания экземпляра любого класса в последующих главах.

В главе рассматриваются следующие темы.

## *Фундаментальная теорема ценообразования финансовых активов*

В этом разделе рассматривается ключевая теорема, которая служит теоретическим обоснованием разрабатываемой библиотеки.

## *Риск-нейтральное дисконтирование*

В этом разделе мы разработаем класс, описывающий риск-нейтральное дисконтирование будущих выплат по опционам и другим производным финансовым инструментам.

## *Рыночная среда*

В этом разделе мы разработаем класс, предназначенный для управления рыночными параметрами ценообразования отдельных финансовых инструментов и портфелей, состоящих из множества таких инструментов.

# Фундаментальная теорема ценообразования финансовых активов

Фундаментальная теорема ценообразования финансовых активов является краеугольным камнем и одним из важных достижений современной финансовой математики<sup>2</sup>. Она базируется на понятии *мартингала* — вероятностной меры, которая позволяет устранить дрейф, обусловленный дисконтированными факторами риска. Другими словами, в случае мартингала все факторы риска дрейфуют с безрисковой краткосрочной ставкой, не испытывая влияния никакой другой ставки, предполагающей любую премию за риск сверх безрисковой краткосрочной ставки.

## Простой пример

Рассмотрим простой пример для двух дат — сегодняшней и завтрашней — рискованного актива (условная акция) и безрискового актива (условная облигация). Облигация сегодня стоит 10 долларов, и завтрашние выплаты тоже составляют 10 долларов (нулевая процентная ставка). Акция сегодня стоит 10 долларов, но завтра она может стоить 20 либо 0 долларов с вероятностью 60% и 40% соответственно. Таким образом, безрисковая доходность облигации равна нулю, а ожидаемая доходность акции равна  $\frac{0,6 \cdot 20 + 0,4 \cdot 0}{10} - 1 = 0,2$ , или 20%. Это и есть *премия за риск* по акции.

Теперь рассмотрим колл-опцион с ценой исполнения 15 долларов. Какой будет его справедливая премия, если известно, что с вероятностью 60% по нему будет выплачено 5 долларов, а в остальных случаях выплата составит 0 долларов? Можно вычислить математическое ожидание и дисконтировать его (в данном случае по нулевой процентной ставке). В результате получаем премию  $0,6 \cdot 5 = 3$  доллара, поскольку выплата 5 долларов по опциону осуществляется только в случае повышения стоимости акции до 20 долларов, а в остальных случаях выплата равна нулю.

Однако существует и другой, не менее эффективный подход к оценке опционов: *репликация* премии опциона через портфель торгуемых ценных бумаг. Нетрудно убедиться, что покупка 0,25 акции позволяет в точности воспроизвести выплаты по опциону (с 60-процентной вероятностью будет получен доход  $0,25 \cdot 20 = 5$  долларов). Вот только четверть акции стоит 2,5, а не 3 доллара.

---

<sup>2</sup> Подробное математическое доказательство этой финансовой теоремы, приведено в книге Делбаена и Шахермайера [4].

Таким образом, учет математического ожидания в условиях реальных рыночных вероятностей приводит к *переоценке* опциона.

Почему так происходит? Реальная вероятностная мера подразумевает премию за риск по акции в размере 20%, поскольку этот риск (заработать 100% или потерять 100%) “реален” в том смысле, что его нельзя диверсифицировать или хеджировать. С другой стороны, премию опциона можно без всякого риска реплицировать с помощью доступного портфеля. Это также подразумевает, что трейдер, выписывающий (продающий) такой опцион, может полностью хеджировать любой риск<sup>3</sup>. Во избежание *арбитража* (ненулевой вероятности получения прибыли в отсутствие дополнительных инвестиций) такой идеально хеджированный портфель, включающий опцион и хеджированную позицию, требует безрисковой ставки.

А можно ли все-таки оценивать колл-опцион на основе математического ожидания? Можно, но только при одном условии: вероятности должны быть пересчитаны таким образом, чтобы рисковый актив (акция) дрейфовал с нулевой безрисковой краткосрочной ставкой. Очевидно, что такое возможно только в случае мартингала, при котором вероятность обоих сценариев равна 50%, т.е.  $\frac{0,5 \cdot 20 + 0,5 \cdot 0}{10} - 1 = 0$ . Тогда расчет математического ожидания выплат по опциону позволяет получить корректную (безарбитражную) справедливую премию опциона:  $0,5 \cdot 5 + 0,5 \cdot 0 = 2,5$ .

## Общая модель

Преимущество описанного выше подхода в том, что его можно распространить даже на самые сложные модели, например с непрерывным временем (когда необходимо учитывать непрерывное множество временных точек), с большим количеством рисковых активов, со сложными выплатами и т.п.

С учетом вышесказанного рассмотрим общую модель финансового рынка с дискретным временем<sup>4</sup>.

*Общая модель финансового рынка  $\mathcal{M}$  с дискретным временем описывается следующими элементами:*

- конечное пространство состояний  $\Omega$ ;
- множество фильтров  $\mathbb{F}$ ;

<sup>3</sup> Инвестиционная стратегия в данном случае предполагает продажу опциона за 2,5 доллара и покупку 0,25 акции за 2,5 доллара. Выплаты по такому портфелю равны нулю независимо от того, какой сценарий осуществится в нашем простом примере.

<sup>4</sup> Принципы расчета вероятностей в такой модели рассмотрены в книге Уильямса [9].

- строго положительная вероятностная мера  $P$ , определенная на множестве  $\wp(\Omega)$ ;
- конечная дата  $T \in \mathbb{N}$ ,  $T < \infty$ ;
- множество  $\mathbb{S} \equiv \{S_t^k\}_{t \in \{0, \dots, T\}} : k \in \{0, \dots, K\}$ , содержащее  $K + 1$  строго положительных процессов оценки стоимости ценных бумаг.

Такую модель можно представить как коллекцию:

$$\mathcal{M} = \{(\Omega, \wp(\Omega), \mathbb{F}, P), T, \mathbb{S}\}.$$

Для этой модели фундаментальная теорема ценообразования финансовых активов формулируется следующим образом<sup>5</sup>.

Рассмотрим общую модель финансового рынка  $\mathcal{M}$ . Согласно фундаментальной теореме ценообразования финансовых активов следующие три утверждения эквивалентны:

- в модели финансового рынка  $\mathcal{M}$  отсутствует возможность арбитражной торговли;
- множество  $\mathbb{Q}$   $P$ -эквивалентных мартингалов не может быть пустым;
- множество  $\mathbb{P}$  согласованных систем линейных уравнений, описывающих стоимость активов, не может быть пустым.

О важности фундаментальной теоремы ценообразования финансовых активов для оценки стоимости деривативов (опционов, фьючерсов, форвардов, свопов и т.п.) можно судить по следующему утверждению.

Если модель финансового рынка  $\mathcal{M}$  является безарбитражной, то у каждого достижимого (т.е. реплицируемого) условного требования (дериватива)  $V_T$  существует уникальная цена  $V_0$ , такая, что  $\forall Q \in \mathbb{Q}: V_0 = E_0^Q(e^{-rT} V_T)$ , где  $e^{-rT}$  — соответствующий риск-нейтральный коэффициент дисконтирования для постоянной краткосрочной ставки  $r$ .

Это не только иллюстрирует важность теоремы, но и показывает, что наши исходные простые рассуждения распространяются на всю модель финансового рынка.

Исходя из роли мартингалов, описанный выше подход к оценке финансовых активов часто называют *мартингальным* или — в связи с тем что мартингал предполагает дрейф всех рисковых активов с безрисковой краткосрочной ставкой — *риск-нейтральным*. Второе название лучше подходит для наших

<sup>5</sup> См. книгу Делбаена и Шахермайера [4].

целей, поскольку в финансовых приложениях часто допускают “упрощение”, позволяя факторам риска (случайным процессам) дрейфовать с безрисковой краткосрочной процентной ставкой. В таких приложениях редко приходится напрямую иметь дело с вероятностными мерами, просто именно они объясняют теоретические результаты и служат основанием для применяемого технического подхода.

Наконец, рассмотрим вопрос полноты финансового рынка, описываемого общей моделью.

Модель рынка  $\mathcal{M}$  является *полной*, если отсутствует арбитраж и любое условное требование (дериватив) достижимо (реплицируемо).

Предположим, модель  $\mathcal{M}$  безарбитражная. Она будет полной тогда и только тогда, когда она единственно возможная, т.е. существует уникальный  $P$ -эквивалентный мартингал.

На этом мы закончим обсуждение теоретических вопросов. Все они детально описаны в книге Хилпиша [6].

## Риск-нейтральное дисконтирование

По вполне понятным причинам риск-нейтральное дисконтирование составляет основу риск-нейтрального подхода. В этом разделе мы разработаем специальный класс Python именно для таких задач. Но сначала имеет смысл разобраться, как моделировать и обрабатывать *релевантные даты* для оценки опционов.

### Моделирование и обработка дат

Прежде чем заниматься дисконтированием, необходимо решить вопрос с моделированием дат (см. приложение А). Для задач оценки опционов необходимо разбить временной отрезок между сегодняшней и конечной датой  $T$  нашей модели рынка на дискретные интервалы, которые могут быть как равномерными (одинаковой длины), так и неравномерными (разной длины). Разрабатываемая нами библиотека должна позволять работать с неравномерными интервалами, тогда более простой случай равномерных интервалов будет обрабатываться автоматически. В приведенном далее коде предполагается, что наименьший интервал времени, на который разбивается диапазон дат, равен *одному дню*. Это означает, что внутридневные операции считаются



несущественными. Для работы с ними необходимо моделировать не только даты, но и время<sup>6</sup>.

Сформировать список релевантных дат можно одним из двух способов: в виде *точных дат* (объектов `datetime`) или в виде *долей года* (как часто делается в теоретических работах).

Для начала импортируем все необходимые библиотеки.

```
In [1]: import numpy as np
import pandas as pd
import datetime as dt
```

```
In [2]: from pylab import mpl, plt
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

```
In [3]: import sys
sys.path.append('../dx')
```

Следующие два списка — `dates` и `fractions` — примерно эквивалентны.

```
In [4]: dates = [dt.datetime(2020, 1, 1), dt.datetime(2020, 7, 1),
dt.datetime(2021, 1, 1)]
```

```
In [5]: (dates[1] - dates[0]).days / 365.
Out[5]: 0.4986301369863014
```

```
In [6]: (dates[2] - dates[1]).days / 365.
Out[6]: 0.5041095890410959
```

```
In [7]: fractions = [0.0, 0.5, 1.0]
```

Мы говорим о *примерной* эквивалентности, поскольку дольные значения редко попадают на начало суток.

Иногда требуется определить доли года по заданному списку дат. Эту задачу решает функция `get_year_deltas()`.

```
#
# Пакет DX
#
# Базовые инструменты -- вспомогательная функция
#
```

---

<sup>6</sup> Добавить компонент времени совсем несложно, но здесь это не сделано, чтобы не загромождать код.

```

# get_year_deltas.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

def get_year_deltas(date_list, day_count=365.):
    ''' Возвращает вектор вещественных чисел,
        представляющих количество дней в долях года.
        Начальное значение нормализуется нулевой датой.

    Параметры
    =====
    date_list: список или массив
               Коллекция объектов datetime
    day_count: float
               Количество дней в году
               (чтобы учесть возможные варианты)

    Результаты
    =====
    delta_list: массив
               Доли года
    ...

    start = date_list[0]
    delta_list = [(date - start).days / day_count
                  for date in date_list]
    return np.array(delta_list)

```

Функция применяется следующим образом.

```
In [8]: from get_year_deltas import get_year_deltas
```

```
In [9]: get_year_deltas(dates)
```

```
Out[9]: array([0.          , 0.49863014, 1.00273973])
```

Преимущества такого преобразования станут очевидными при моделировании краткосрочной ставки.

## Постоянная краткосрочная ставка

В следующем примере мы рассмотрим самый простой вариант дисконтирования с краткосрочной процентной ставкой, при котором она остается *постоянной во времени*. На этом предположении<sup>7</sup> основаны многие модели оценки опционов, например модели Блэка — Шоулза — Мертона ([2] и [7]), Мертона [8] и Кокса — Росса — Рубинштейна [3]. В случае непрерывного дисконтирования, которое обычно имеет место при оценке опционов, общий коэффициент дисконтирования применительно к сегодняшней дате выражается через будущую дату  $t$  и постоянную краткосрочную ставку  $r$  по формуле  $D_0(t) = e^{-rt}$ . В конце периода дисконтирования получаем  $D_0(T) = e^{-rT}$ . Запомним, что величины  $t$  и  $T$  представляются долями года.

Коэффициенты дисконтирования можно также интерпретировать как текущую стоимость *единичной дисконтной облигации* (Zero Coupon Bond — ZCB) со сроком погашения в моменты времени  $t$  или  $T$  соответственно<sup>8</sup>. Коэффициент дисконтирования для дат  $t$  и  $s$ , где  $t \geq s \geq 0$ , вычисляется так:

$$D_s(t) = \frac{D_0(t)}{D_0(s)} = \frac{e^{-rt}}{e^{-rs}} = e^{-rt} e^{rs} = e^{-r(t-s)}.$$

Реализуем все это в виде класса Python<sup>9</sup>.

```
#
# Пакет DX
#
# Базовые инструменты -- класс постоянной краткосрочной ставки
#
# constant_short_rate.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
```

---

<sup>7</sup> Оно оказывается вполне справедливым при оценке, например, краткосрочных опционов.

<sup>8</sup> Единичная дисконтная облигация предполагает выплату ровно одной денежной единицы в момент погашения и не имеет купонов между сегодняшней датой и датой исполнения.

<sup>9</sup> Основы объектно-ориентированного программирования на Python рассматривались в главе 6. В этой и последующих главах имена классов не соответствуют стандартному стилю PEP 8, в котором рекомендуется придерживаться “верблюжьего регистра” (CamelCase). Мы будем использовать “змеиный регистр” (snake\_case), более характерный для *имен функций*. В руководстве PEP 8 он допускается как альтернативный вариант “в случаях, когда интерфейс задокументирован и используется в основном для вызова методов”.

```

from get_year_deltas import *

class constant_short_rate(object):
    ''' Класс дисконтирования с постоянной краткосрочной ставкой

    Атрибуты
    =====
    name: строка
           Имя объекта
    short_rate: float (положительное число)
           Постоянная ставка дисконтирования

    Методы
    =====
    get_discount_factors:
           Возвращает коэффициенты дисконтирования для заданного
           списка/массива объектов datetime или долей года
    '''

    def __init__(self, name, short_rate):
        self.name = name
        self.short_rate = short_rate
        if short_rate < 0:
            raise ValueError('Short rate negative.')

    def get_discount_factors(self, date_list, dtobjects=True):
        if dtobjects is True:
            dlist = get_year_deltas(date_list)
        else:
            dlist = np.array(date_list)
        dflist = np.exp(self.short_rate * np.sort(-dlist))
        return np.array((date_list, dflist)).T

```

Применение класса `dx.constant_short_rate` лучше всего проиллюстрировать на простом примере. В данном случае мы получаем двухмерный массив `ndarray`, содержащий объекты `datetime` и соответствующие им коэффициенты дисконтирования. Класс также поддерживает работу с долями года.

```
In [10]: from constant_short_rate import constant_short_rate
```

```
In [11]: csr = constant_short_rate('csr', 0.05)
```

```
In [12]: csr.get_discount_factors(dates)
```

```

Out[12]: array([[datetime.datetime(2020, 1, 1, 0, 0), 0.9510991280247174],
               [datetime.datetime(2020, 7, 1, 0, 0), 0.9753767163648953],
               [datetime.datetime(2021, 1, 1, 0, 0), 1.0]], dtype=object)

In [13]: deltas = get_year_deltas(dates)
          deltas
Out[13]: array([0.          , 0.49863014, 1.00273973])

In [14]: csr.get_discount_factors(deltas, dtobjects=False)
Out[14]: array([[0.          , 0.95109913],
               [0.49863014, 0.97537672],
               [1.00273973, 1.          ]])

```

С помощью этого класса выполняются все операции дисконтирования, требуемые в других классах.

## Рыночная среда

Под *рыночной средой* мы будем понимать коллекцию вспомогательных данных и объектов Python, используемых в финансовых расчетах. Наличие подобной абстракции упрощает выполнение ряда операций и позволяет согласованно моделировать повторяющиеся элементы<sup>10</sup>. Рыночная среда состоит из трех словарей, предназначенных для хранения следующих типов данных и объектов Python.

### *Константы*

Это могут быть, например, параметры модели или даты исполнения опционов.

### *Списки*

Это общие коллекции объектов, например список объектов, моделирующих поведение (рисковых) ценных бумаг.

### *Кривые*

Это объекты для дисконтирования, например экземпляры класса `dx.constant_short_rate`.

Ниже приведен код класса `dx.market_environment` (работа со словарями рассматривалась в главе 3).

---

<sup>10</sup> См. книгу Флетчера и Гарднера [5], в которой эта концепция широко применяется.

```

#
# Пакет DX
#
# Базовые инструменты -- класс рыночной среды
#
# market_environment.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#

class market_environment(object):
    ''' Класс моделирования рыночной среды, в которой
        выполняется оценка финансовых инструментов.

        Атрибуты
        =====
        name: строка
            Имя рыночной среды
        pricing_date: объект datetime
            Дата рыночной среды

        Методы
        =====
        add_constant:
            Добавляет константу (например, параметр модели)
        get_constant:
            Возвращает константу
        add_list:
            Добавляет список (например, базовых активов)
        get_list:
            Возвращает список
        add_curve:
            Добавляет рыночную кривую (например, кривую доходности)
        get_curve:
            Возвращает рыночную кривую
        add_environment:
            Добавляет или переписывает всю рыночную
            среду, включая константы, списки и кривые
        ...

    def __init__(self, name, pricing_date):
        self.name = name

```

```

        self.pricing_date = pricing_date
        self.constants = {}
        self.lists = {}
        self.curves = {}

    def add_constant(self, key, constant):
        self.constants[key] = constant

    def get_constant(self, key):
        return self.constants[key]

    def add_list(self, key, list_object):
        self.lists[key] = list_object

    def get_list(self, key):
        return self.lists[key]

    def add_curve(self, key, curve):
        self.curves[key] = curve

    def get_curve(self, key):
        return self.curves[key]

    def add_environment(self, env):
        # Перезапись существующих значений
        self.constants.update(env.constants)
        self.lists.update(env.lists)
        self.curves.update(env.curves)

```

Несмотря на то что в самом классе `dx.market_environment` нет ничего особенного, следующий простой пример наглядно показывает, насколько удобно работать с экземплярами такого класса.

```

In [15]: from market_environment import market_environment

In [16]: me = market_environment('me_gbm', dt.datetime(2020, 1, 1))

In [17]: me.add_constant('initial_value', 36.)

In [18]: me.add_constant('volatility', 0.2)

In [19]: me.add_constant('final_date', dt.datetime(2020, 12, 31))

In [20]: me.add_constant('currency', 'EUR')

```

```
In [21]: me.add_constant('frequency', 'M')

In [22]: me.add_constant('paths', 10000)

In [23]: me.add_curve('discount_curve', csr)

In [24]: me.get_constant('volatility')
Out[24]: 0.2

In [25]: me.get_curve('discount_curve').short_rate
Out[25]: 0.05
```

Здесь продемонстрированы общие принципы применения этого универсального класса, предназначенного для хранения данных. На практике нужно сначала собрать все необходимые рыночные и другие данные, а также объекты Python, и только затем создать объект `dx.market_environment`, заполнив его данными. Впоследствии этот объект можно передавать другим классам, которым требуются хранящиеся в нем данные.

Основное преимущество такого объектно-ориентированного подхода к моделированию заключается в том, что, например, экземпляры класса `dx.constant_short_rate` могут существовать сразу в нескольких средах (см. описание принципов агрегирования в главе 6). При изменении экземпляра (когда, допустим, задается новая постоянная краткосрочная ставка) автоматически обновляются все экземпляры класса `dx.market_environment`, в которых используется этот конкретный экземпляр класса дисконтирования.



### Гибкость класса

Класс рыночной среды, описанный в этом разделе, представляет собой достаточно гибкое средство моделирования и хранения любых параметров и входных данных, связанных с оценкой опционов и их портфелей. В то же время такая гибкость ведет к операционным рискам, поскольку на этапе создания экземпляра класса можно легко передать лишние данные или объекты, которые не будут выявлены. В производственных приложениях необходимо добавлять проверки, позволяющие обнаружить хотя бы наиболее очевидные ошибки.



## Резюме

В этой главе был реализован базовый каркас более крупного проекта по созданию пакета Python, предназначенного для оценки опционов и других деривативов по методу Монте-Карло. Мы рассмотрели фундаментальную теорему ценообразования финансовых активов, проиллюстрировав ее на относительно простом примере, и на ее основе составили общую модель финансового рынка с дискретным временем.

Мы также разработали класс `dx.constant_short_rate`, предназначенный для риск-нейтрального дисконтирования. Он позволяет на основе списка объектов `datetime` либо списка вещественных чисел, представляющих доли года, получить соответствующие коэффициенты дисконтирования (текущие стоимости единичных дисконтных облигаций).

Наконец, мы разработали универсальный класс `dx.market_environment`, который позволяет создавать коллекции всех необходимых параметров и объектов Python, применяемых в процессе моделирования и оценки финансовых инструментов.

Чтобы упростить последующий импорт модулей, мы создадим обобщенный модуль `dx_frame.py`.

```
#
# Пакет DX
#
# Базовые функции и классы
#
# dx_frame.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import datetime as dt

from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
```

Теперь для подключения всех необходимых компонентов достаточно одной-единственной команды импорта:

```
import dx_frame
```

При работе с пакетом Python можно сохранить все необходимые модули в отдельной папке, добавив в нее специальный файл `__init__.py` (обратите внимание на соглашение об именовании), содержащий все команды импорта. Вот как будет выглядеть такой файл, который мы поместим в папку *dx* вместе со всеми модулями.

```
#
# Пакет DX
#
# Пакетный файл
#
# __init__.py
#
import datetime as dt

from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
```

Теперь в инструкции импорта достаточно указать только имя папки:

```
from dx import *
```

Или альтернативный вариант:

```
import dx
```

## Дополнительные ресурсы

В главе упоминались следующие источники.

1. Bittman, James. *Trading Options as a Professional* (2009, McGraw Hill).
2. Black, Fischer, and Myron Scholes. *The Pricing of Options and Corporate Liabilities* (1973, *The Journal of Political Economy*, Vol. 81, No. 3, pp. 637–654).
3. Cox, John, Stephen Ross, and Mark Rubinstein. *Option Pricing: A Simplified Approach* (1979, *Journal of Financial Economics*, Vol. 7, No. 3, pp. 229–263).
4. Delbaen, Freddy, and Walter Schachermayer. *The Mathematics of Arbitrage* (2004, Springer-Verlag).
5. Fletcher, Shayne, and Christopher Gardner. *Financial Modelling in Python* (2009, Wiley).
6. Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).

7. Merton, Robert. *Theory of Rational Option Pricing* (1973, *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141–183).
8. Merton, Robert. *Option Pricing When the Underlying Stock Returns Are Discontinuous* (1976, *Journal of Financial Economics*, Vol. 3, No. 3, pp. 125–144).
9. Williams, David. *Probability with Martingales* (1991, Cambridge University Press).

В последующих главах будут даны ссылки на другие научные публикации, в которых описаны рассмотренные здесь математические модели.

---

# Финансовое моделирование

Цель науки — не анализировать или описывать, а создавать  
подходящие модели мира.

*Эдвард де Бона*

В главе 12 мы уже применяли метод Монте-Карло для моделирования случайных процессов с использованием библиотеки NumPy. В этой главе мы воспользуемся имеющимися наработками для реализации классов моделирования, которые составляют основу пакета DX.

В главе рассматриваются следующие темы.

## *Генерирование случайных чисел*

В этом разделе мы разработаем функцию генерирования случайных чисел со стандартным нормальным распределением, применив технику уменьшения дисперсии<sup>1</sup>.

## *Общий класс моделирования*

В этом разделе мы создадим общий класс моделирования, от которого другие, более специализированные классы будут наследовать основные атрибуты и методы.

## *Геометрическое броуновское движение*

Этот раздел посвящен геометрическому броуновскому движению, которое было впервые применено для оценки опционов Блэком и Шоулзом [1], а также Мертоном [5]. Несмотря на известные недостатки и противоречивые эмпирические результаты, оно все еще остается эталонным методом оценки опционов и других деривативов.

## *Прыжковая диффузия*

Прыжковая диффузия, применительно к финансам впервые описанная Мертоном [6], добавляет к геометрическому броуновскому движению прыжковый компонент с логнормальным распределением. Это, в

---

<sup>1</sup> Всякий раз, говоря о “случайных” числах, мы подразумеваем псевдослучайные числа.

частности, дает возможность учесть тот факт, что краткосрочные опционы “без денег” (out-of-the-money — OTM) часто оцениваются исходя из вероятности больших скачков. Другими словами, моделирование только на основе геометрического броуновского движения не позволяет удовлетворительно объяснить рыночную стоимость таких OTM-опционов, в отличие от модели прыжковой диффузии.

#### *Диффузия по закону квадратного корня*

Этот вид диффузии, предложенный Коксом, Ингерсоллом и Россом [2], применяется для моделирования величин, характеризующихся возвратом к среднему (реверсией), например процентных ставок и волатильности. Кроме того, диффузионный процесс остается строго положительным, что важно при работе с такими величинами.

Более детальное описание представленных в этой главе моделей приведено у Хилпиша [4]. В частности, там рассмотрен конкретный пример применения модели прыжковой диффузии Мертона [6].

## Генерирование случайных чисел

Генерирование случайных чисел — ключевая задача в методе Монте-Карло<sup>2</sup>. В главе 12 было показано, как генерировать случайные числа с разными типами распределений с помощью встроенных средств Python и вспомогательных пакетов, таких как `numpy` и `gandom`. Для нашего проекта основной интерес представляют случайные числа со *стандартным нормальным распределением*. С этой целью мы напишем вспомогательную функцию `sn_gandom_numbers()`, которая генерирует именно такие числа.

```
#
# Пакет DX
#
# Базовые инструменты -- генератор случайных чисел
#
# sn_random_numbers.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
```

---

<sup>2</sup> О генерировании случайных чисел подробно рассказано у Глассермана [3].

```
def sn_random_numbers(shape, antithetic=True,
                      moment_matching=True,
                      fixed_seed=False):
    ''' Возвращает объект ndarray формы shape, содержащий
        (псевдо)случайные числа со стандартным нормальным
        распределением
```

*Параметры*

=====

*shape: кортеж (о, п, т)*

*Создание массива формы (о, п, т)*

*antithetic: булево значение*

*Метод симметричных выборок*

*moment\_matching: булево значение*

*Метод моментов*

*fixed\_seed: булево значение*

*Флаг фиксации заправочного значения*

*Результаты*

=====

*ran: массив (о, п, т) (псевдо)случайных чисел*  
'''

```
if fixed_seed:
    np.random.seed(1000)
if antithetic:
    ran = np.random.standard_normal(
        (shape[0], shape[1], shape[2] // 2))
    ran = np.concatenate((ran, -ran), axis=2)
else:
    ran = np.random.standard_normal(shape)
if moment_matching:
    ran = ran - np.mean(ran)
    ran = ran / np.std(ran)
if shape[0] == 1:
    return ran[0]
else:
    return ran
```

Задействуемые в этой функции способы уменьшения дисперсии, в частности метод симметричных выборок и метод моментов, рассматривались в главе 12<sup>3</sup>. Работать с функцией несложно.

<sup>3</sup> Обзор и теоретическое обоснование различных методов уменьшения дисперсии даны у Гласермана [3].

```
In [26]: from sn_random_numbers import *
```

```
In [27]: snrn = sn_random_numbers((2, 2, 2), antithetic=False,  
                                   moment_matching=False,  
                                   fixed_seed=True)
```

```
snrn
```

```
Out[27]: array([[[-0.8044583 , 0.32093155],  
                 [-0.02548288, 0.64432383]],  
                [[-0.30079667, 0.38947455],  
                 [-0.1074373 , -0.47998308]]])
```

```
In [28]: round(snrn.mean(), 6)
```

```
Out[28]: -0.045429
```

```
In [29]: round(snrn.std(), 6)
```

```
Out[29]: 0.451876
```

```
In [30]: snrn = sn_random_numbers((2, 2, 2), antithetic=False,  
                                   moment_matching=True,  
                                   fixed_seed=True)
```

```
snrn
```

```
Out[30]: array([[[-1.67972865, 0.81075283],  
                 [ 0.04413963, 1.52641815]],  
                [[-0.56512826, 0.96243813],  
                 [-0.13722505, -0.96166678]]])
```

```
In [31]: round(snrn.mean(), 6)
```

```
Out[31]: -0.0
```

```
In [32]: round(snrn.std(), 6)
```

```
Out[32]: 1.0
```

Данная функция будет активно применяться в создаваемых далее классах моделирования.

## Общий класс моделирования

Объектно-ориентированная парадигма (см. главу 6) позволяет наследовать атрибуты и методы классов. Мы воспользуемся этим при разработке классов моделирования: сначала мы напишем *общий* класс, содержащий

универсальные для всех классов атрибуты и методы, а затем сосредоточимся на моделировании более специфических элементов в других классах.

Конструктору любого класса моделирования необходимо передать три атрибута.

`name`

Имя объекта модели.

`mag_env`

Экземпляр класса `dx.market_environment`.

`corr`

Булев флаг, указывающий на наличие или отсутствие корреляции.

Этот подход еще раз подчеркивает важность концепции *рыночной среды*, которая позволяет за один заход предоставить все необходимые данные и объекты. Общий класс включает следующие методы.

`generate_time_grid()`

Генерирует сетку релевантных дат для моделирования. Эта операция потребуется во всех классах.

`get_instrument_values()`

Каждый класс должен возвращать объект `pd.DataFrame`, хранящий значения моделируемого инструмента (например, котировки акций, цены на биржевые товары, волатильности и т.п.).

Ниже показан код общего класса. В нем вызываются методы, предоставляемые другими классами моделирования, например `self.generate_paths()`. Их назначение станет понятно позже, когда мы займемся разработкой специализированных классов.

```
#
# Пакет DX
#
# Класс моделирования -- базовый класс
#
# simulation_class.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd
```



```

class simulation_class(object):
    ''' Реализует базовые методы для других классов моделирования

    Атрибуты
    =====
    name: строка
        Имя объекта
    mar_env: экземпляр класса market_environment
        Рыночная среда для моделирования
    corr: булево значение
        True, если есть корреляция с другим объектом

    Методы
    =====
    generate_time_grid:
        Возвращает сетку дат для моделирования
    get_instrument_values:
        Возвращает текущие значения инструмента (массив)
    '''

    def __init__(self, name, mar_env, corr):
        self.name = name
        self.pricing_date = mar_env.pricing_date
        self.initial_value = mar_env.get_constant('initial_value')
        self.volatility = mar_env.get_constant('volatility')
        self.final_date = mar_env.get_constant('final_date')
        self.currency = mar_env.get_constant('currency')
        self.frequency = mar_env.get_constant('frequency')
        self.paths = mar_env.get_constant('paths')
        self.discount_curve = mar_env.get_curve('discount_curve')
        try:
            # Если объект time_grid включен в коллекцию mar_env,
            # используем его (для оценки портфеля)
            self.time_grid = mar_env.get_list('time_grid')
        except:
            self.time_grid = None
        try:
            # Если имеются специальные даты, добавляем их
            self.special_dates = mar_env.get_list('special_dates')
        except:
            self.special_dates = []
        self.instrument_values = None

```

```

self.correlated = corr
if corr is True:
    # Требуется только в контексте портфеля
    # с коррелированными факторами риска
    self.cholesky_matrix = mar_env.get_list('cholesky_matrix')
    self.rn_set = mar_env.get_list('rn_set')[self.name]
    self.random_numbers = mar_env.get_list('random_numbers')

def generate_time_grid(self):
    start = self.pricing_date
    end = self.final_date
    # Функция date_range() библиотеки pandas;
    # параметр freq может быть равен, например,
    # 'B' (рабочие дни), 'W' (еженедельно), 'M' (ежемесячно)
    time_grid = pd.date_range(start=start, end=end,
                              freq=self.frequency).to_pydatetime()
    time_grid = list(time_grid)
    # Дополнение объекта time_grid значениями start,
    # end и special_dates
    if start not in time_grid:
        # Добавление начальной даты, если ее нет в списке
        time_grid.insert(0, start)
    if end not in time_grid:
        # Добавление конечной даты, если ее нет в списке
        time_grid.append(end)
    if len(self.special_dates) > 0:
        # Добавление всех специальных дат
        time_grid.extend(self.special_dates)
        # Удаление повторов
        time_grid = list(set(time_grid))
        # Сортировка списка
        time_grid.sort()
    self.time_grid = np.array(time_grid)

def get_instrument_values(self, fixed_seed=True):
    if self.instrument_values is None:
        # Запускать моделирование, только если
        # нет значений инструментов
        self.generate_paths(fixed_seed=fixed_seed,
                           day_count=365.)
    elif fixed_seed is False:
        # Запускать повторное моделирование,
        # если параметр fixed_seed равен False

```

```

        self.generate_paths(fixed_seed=fixed_seed,
                             day_count=365.)
    return self.instrument_values

```

Анализ рыночной среды выполняется в специальном методе `__init__()`, который вызывается при создании объекта. Чтобы не загромождать код, мы *не* проверяем корректность предоставленных значений. Например, следующая строка кода успешно выполняется независимо от того, действительно ли в коллекции хранится экземпляр класса дисконтирования:

```
self.discount_curve = mar_env.get_curve('discount_curve')
```

Поэтому нужно быть очень осторожным при передаче объектов `dx.market_environment` в любой класс моделирования.

В табл. 18.1 перечислены все компоненты, которые должен содержать объект `dx.market_environment`, передаваемый общему классу моделирования (а значит, и всем остальным классам).

*Таблица 18.1. Элементы рыночной среды, предоставляемые для всех классов моделирования*

Элемент	Тип	Обязательный	Описание
<code>initial_value</code>	Константа	Да	Начальное значение процесса на дату <code>pricing_date</code>
<code>volatility</code>	Константа	Да	Коэффициент волатильности процесса
<code>final_date</code>	Константа	Да	Горизонт моделирования
<code>currency</code>	Константа	Да	Валюта финансового инструмента
<code>frequency</code>	Константа	Да	Периодичность выборки данных, используемая в качестве параметра <code>freq</code> библиотеки <code>pandas</code>
<code>paths</code>	Константа	Да	Количество моделируемых траекторий
<code>discount_curve</code>	Кривая	Да	Экземпляр класса <code>dx.constant_short_rate</code>
<code>time_grid</code>	Список	Нет	Сетка анализируемого диапазона дат (в контексте портфеля)
<code>random_numbers</code>	Список	Нет	Объект <code>ndarray</code> , содержащий случайные числа (для коррелированных объектов)
<code>cholesky_matrix</code>	Список	Нет	Матрица Холецкого (для коррелированных объектов)
<code>rn_set</code>	Список	Нет	Словарь указателей на соответствующие наборы случайных чисел

Все, что связано с корреляцией моделируемых объектов, будет объясняться в последующих главах. В этой главе мы концентрируемся на моделировании одиночных процессов, не подверженных корреляции. То же самое касается объекта `time_grid`, который нужен только в контексте портфеля (эта тема рассматривается в последующих главах).

## Геометрическое броуновское движение

Геометрическое броуновское движение — это случайный процесс, описываемый уравнением 18.1 (см. также уравнение 12.2). Дрейф процесса здесь уже задан равным безрисковой постоянной краткосрочной ставке  $r$ , а значит, перед нами мартингал (см. главу 17).

*Уравнение 18.1. Стохастическое дифференциальное уравнение геометрического броуновского движения*

$$dS_t = rS_t dt + \sigma S_t dZ.$$

Уравнение 18.2 представляет собой метод Эйлера (см. также уравнение 12.3), который мы будем применять для целей моделирования. Основой здесь служит модель рынка с дискретным временем, например общая модель финансового рынка  $\mathcal{M}$  (см. главу 17) с конечным диапазоном рассматриваемых дат  $0 < t_1 < t_2 < \dots < T$ .

*Уравнение 18.2. Рекуррентное уравнение для моделирования геометрического броуновского движения*

$$S_{t_{m+1}} = S_{t_m} \exp \left( \left( r - \frac{\sigma^2}{2} \right) (t_{m+1} - t_m) + \sigma \sqrt{t_{m+1} - t_m} z_t \right), \quad 0 \leq t_m \leq t_{m+1} \leq T.$$

## Класс моделирования

Ниже приведен код класса, моделирующего геометрическое броуновское движение.

```
#
# Пакет DX
#
# Класс моделирования -- геометрическое броуновское движение
#
# geometric_brownian_motion.py
#
```

```

# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class

class geometric_brownian_motion(simulation_class):
    ''' Класс, генерирующий траектории геометрического броуновского
        движения в соответствии с моделью Блэка – Шоулза – Мертона

    Атрибуты
    =====
    name: строка
           Имя объекта
    mar_env: экземпляр класса market_environment
           Рыночная среда для моделирования
    corr: булево значение
           True, если есть корреляция с другим объектом

    Методы
    =====
    update:
           Обновление параметров
    generate_paths:
           Возвращает траектории, рассчитанные по методу
           Монте-Карло для заданной рыночной среды
    ...

    def __init__(self, name, mar_env, corr=False):
        super(geometric_brownian_motion, self).__init__(name, mar_env,
                                                         corr)

    def update(self, initial_value=None, volatility=None,
               final_date=None):
        if initial_value is not None:
            self.initial_value = initial_value
        if volatility is not None:
            self.volatility = volatility
        if final_date is not None:
            self.final_date = final_date
        self.instrument_values = None

```

```

def generate_paths(self, fixed_seed=False, day_count=365.):
    if self.time_grid is None:
        # Метод, заимствованный у общего класса моделирования
        self.generate_time_grid()
    # Количество дат в сетке
    M = len(self.time_grid)
    # Количество траекторий
    I = self.paths
    # Инициализация объекта ndarray для моделирования траекторий
    paths = np.zeros((M, I))
    # Инициализация первой даты значением initial_value
    paths[0] = self.initial_value
    if not self.correlated:
        # При отсутствии корреляции генерируются
        # случайные числа
        rand = sn_random_numbers((1, M, I),
                                fixed_seed=fixed_seed)
    else:
        # При наличии корреляции используется объект случайных
        # чисел, предоставляемый рыночной средой
        rand = self.random_numbers
    # Получение краткосрочной ставки для учета дрейфа процесса
    short_rate = self.discount_curve.short_rate
    for t in range(1, len(self.time_grid)):
        # Выбор подходящего среза из набора случайных чисел
        if not self.correlated:
            ran = rand[t]
        else:
            ran = np.dot(self.cholesky_matrix, rand[:, t, :])
            ran = ran[self.rn_set]
        # Разница между двумя датами, выраженная в долях года
        dt = (self.time_grid[t] - self.time_grid[t - 1]).days /
            day_count
        # Генерирование значений для соответствующей даты
        paths[t] = paths[t - 1] * np.exp((short_rate - 0.5 *
            self.volatility ** 2) * dt +
            self.volatility * np.sqrt(dt) * ran)
    self.instrument_values = paths

```

В данном конкретном случае объект `dx.market_environment` должен содержать только даты и объекты, перечисленные в табл. 18.1 (т.е. минимальный набор компонентов).

Метод `update()` достаточно прост: его задача — обновить ключевые параметры модели. А вот метод `generate_paths()` гораздо сложнее, поэтому все его инструкции прокомментированы в коде. Определенная сложность объясняется необходимостью учесть корреляцию между различными объектами модели (об этом мы поговорим в главе 20).

## Пример использования

В следующем интерактивном сеансе IPython показано применение класса, моделирующего геометрическое броуновское движение. Сначала необходимо создать объект рыночной среды `dx.market_environment`, содержащий все обязательные элементы.

```
In [33]: from dx_frame import *
```

```
In [34]: me_gbm = market_environment('me_gbm',  
                                     dt.datetime(2020, 1, 1))
```

```
In [35]: me_gbm.add_constant('initial_value', 36.)  
me_gbm.add_constant('volatility', 0.2)  
me_gbm.add_constant('final_date', dt.datetime(2020, 12, 31))  
me_gbm.add_constant('currency', 'EUR')  
me_gbm.add_constant('frequency', 'M') ❶  
me_gbm.add_constant('paths', 10000)
```

```
In [36]: csr = constant_short_rate('csr', 0.06)
```

```
In [37]: me_gbm.add_curve('discount_curve', csr)
```

❶ Ежемесячная периодичность с окончанием интервалов в конце месяца (по умолчанию)

Далее создается объект модели.

```
In [38]: from geometric_brownian_motion  
import geometric_brownian_motion
```

```
In [39]: gbm = geometric_brownian_motion('gbm', me_gbm) ❷
```

```
In [40]: gbm.generate_time_grid() ❸
```

```
In [41]: gbm.time_grid ❹
```

```
Out[41]: array([datetime.datetime(2020, 1, 1, 0, 0),  
               datetime.datetime(2020, 1, 31, 0, 0),
```

```
datetime.datetime(2020, 2, 29, 0, 0),
datetime.datetime(2020, 3, 31, 0, 0),
datetime.datetime(2020, 4, 30, 0, 0),
datetime.datetime(2020, 5, 31, 0, 0),
datetime.datetime(2020, 6, 30, 0, 0),
datetime.datetime(2020, 7, 31, 0, 0),
datetime.datetime(2020, 8, 31, 0, 0),
datetime.datetime(2020, 9, 30, 0, 0),
datetime.datetime(2020, 10, 31, 0, 0),
datetime.datetime(2020, 11, 30, 0, 0),
datetime.datetime(2020, 12, 31, 0, 0)], dtype=object)
```

```
In [42]: %time paths_1 = gbm.get_instrument_values() ④
CPU times: user 21.3 ms, sys: 6.74 ms, total: 28.1 ms
Wall time: 40.3 ms
```

```
In [43]: paths_1.round(3) ④
Out[43]: array([[36.    , 36.    , 36.    , ..., 36.    , 36.    , 36.    ],
 [37.403, 38.12  , 34.4   , ..., 36.252, 35.084, 39.668],
 [39.562, 42.335, 32.405, ..., 34.836, 33.637, 37.655],
 ...,
 [40.534, 33.506, 23.497, ..., 37.851, 30.122, 30.446],
 [42.527, 36.995, 21.885, ..., 36.014, 30.907, 30.712],
 [43.811, 37.876, 24.1   , ..., 36.263, 28.138, 29.038]])
```

```
In [44]: gbm.update(volatility=0.5) ⑤
```

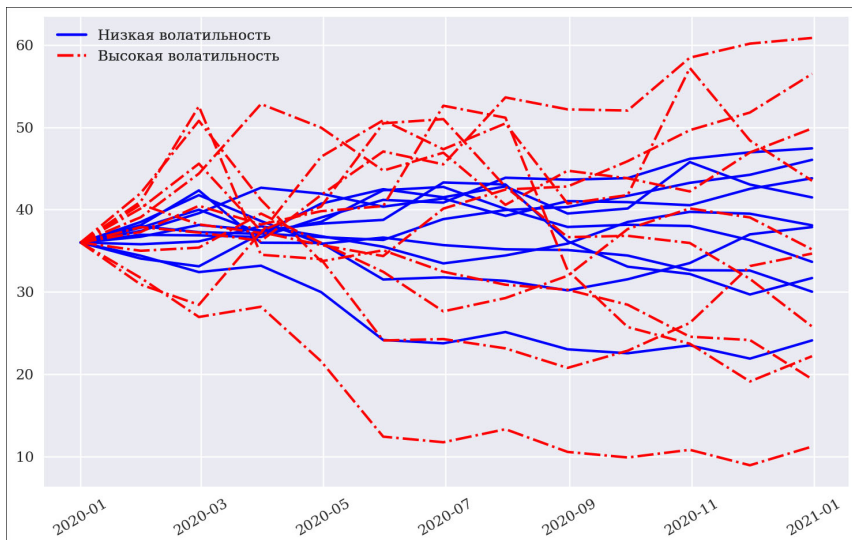
```
In [45]: %time paths_2 = gbm.get_instrument_values() ⑤
CPU times: user 27.8 ms, sys: 3.91 ms, total: 31.7 ms
Wall time: 19.8 ms
```

- ① Инициализация объекта моделирования.
- ② Генерирование сетки дат...
- ③ ...и ее отображение. Обратите внимание на добавление начальной даты.
- ④ Расчет траекторий с учетом заданных параметров.
- ⑤ Обновление параметра волатильности и повторное моделирование.

На рис. 18.1 показан результат расчета десяти траекторий при двух разных вариантах параметризации. Несложно заметить, к чему приводит увеличение волатильности.



```
In [46]: plt.figure(figsize=(10, 6))
p1 = plt.plot(gbm.time_grid, paths_1[:, :10], 'b')
p2 = plt.plot(gbm.time_grid, paths_2[:, :10], 'r-.')
l1 = plt.legend([p1[0], p2[0]],
                ['Низкая волатильность',
                 'Высокая волатильность'],
                loc=2)
plt.gca().add_artist(l1)
plt.xticks(rotation=30);
```



*Рис. 18.1. Смоделированные траектории геометрического броуновского движения*



### Векторизация моделирования

Как было показано в главе 12, векторизация, выполняемая средствами библиотек NumPy и pandas, позволяет получить более компактный и производительный код.

## Прыжковая диффузия

После класса `dx.geometric_brownian_motion` нам не составит особого труда реализовать класс для модели прыжковой диффузии, описанной у Мертона [6]. Ниже приведено стохастическое дифференциальное уравнение, описывающее прыжковую диффузию (см. также уравнение 12.8).

*Уравнение 18.3. Стохастическое дифференциальное уравнение для модели прыжковой диффузии Мертона*

$$dS_t = (r - r_f) S_t dt + \sigma S_t dZ_t + J_t S_t dN_t.$$

Метод Эйлера, применяемый для задач моделирования, представлен уравнением 18.4 (см. также уравнение 12.9).

*Уравнение 18.4. Метод Эйлера для дискретизации модели прыжковой диффузии Мертона*

$$S_{t_{m+1}} = S_{t_m} \left( \exp \left( \left( r - r_f - \frac{\sigma^2}{2} \right) (t_{m+1} - t_m) + \sigma \sqrt{t_{m+1} - t_m} z_t^1 \right) + \left( e^{\mu_J + \delta z_t^2} - 1 \right) y_t \right),$$
$$0 \leq t_m \leq t_{m+1} \leq T.$$

## Класс моделирования

Ниже показан код класса `dx.jump_diffusion`. Он уже не должен вызывать у вас никаких вопросов. Понятно, что модель здесь иная, но сама структура класса и его методы те же самые.

```
#
# Пакет DX
#
# Класс моделирования -- прыжковая диффузия
#
# jump_diffusion.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
```

```

class jump_diffusion(simulation_class):
    ''' Класс, генерирующий траектории на основе
        модели прыжковой диффузии Мертона

    Атрибуты
    =====
    name: строка
           Имя объекта
    mar_env: экземпляр класса market_environment
           Рыночная среда для моделирования
    corr: булево значение
           True, если есть корреляция с другим объектом

    Методы
    =====
    update:
           Обновление параметров
    generate_paths:
           Возвращает траектории, рассчитанные по методу
            Монте-Карло для заданной рыночной среды
    ...

    def __init__(self, name, mar_env, corr=False):
        super(jump_diffusion, self).__init__(name, mar_env, corr)
        # Требуются дополнительные параметры
        self.lamb = mar_env.get_constant('lambda')
        self.mu = mar_env.get_constant('mu')
        self.delt = mar_env.get_constant('delta')

    def update(self, initial_value=None, volatility=None,
               lamb=None, mu=None, delta=None, final_date=None):
        if initial_value is not None:
            self.initial_value = initial_value
        if volatility is not None:
            self.volatility = volatility
        if lamb is not None:
            self.lamb = lamb
        if mu is not None:
            self.mu = mu
        if delta is not None:
            self.delt = delta
        if final_date is not None:

```

```

        self.final_date = final_date
        self.instrument_values = None

def generate_paths(self, fixed_seed=False, day_count=365.):
    if self.time_grid is None:
        # Метод, заимствованный у общего класса моделирования
        self.generate_time_grid()
    # Количество дат в сетке
    M = len(self.time_grid)
    # Количество траекторий
    I = self.paths
    # Инициализация объекта ndarray, для моделирования траекторий
    paths = np.zeros((M, I))
    # Инициализация первой даты значением initial_value
    paths[0] = self.initial_value
    if self.correlated is False:
        # При отсутствии корреляции генерируются
        # случайные числа
        sn1 = sn_random_numbers((1, M, I), fixed_seed=fixed_seed)
    else:
        # При наличии корреляции используется объект случайных
        # чисел, предоставляемый рыночной средой
        sn1 = self.random_numbers

    # Псевдослучайные числа со стандартным нормальным
    # распределением, описывающие прыжковую компоненту модели
    sn2 = sn_random_numbers((1, M, I), fixed_seed=fixed_seed)

    rj = self.lamb * (np.exp(self.mu + 0.5 * self.delt ** 2) - 1)

    short_rate = self.discount_curve.short_rate
    for t in range(1, len(self.time_grid)):
        # Выбор подходящего среза из набора случайных чисел
        if self.correlated is False:
            ran = sn1[t]
        else:
            # Только в случае корреляции в контексте портфеля
            ran = np.dot(self.cholesky_matrix, sn1[:, t, :])
            ran = ran[self.rn_set]
        # Разница между двумя датами, выраженная в долях года
        dt = (self.time_grid[t] - self.time_grid[t - 1]).days /
            day_count

```

```

# Псевдослучайные числа с распределением Пуассона,
# описывающие прыжковую компоненту модели
poi = np.random.poisson(self.lamb * dt, 1)
paths[t] = paths[t - 1] * (
    np.exp((short_rate - rj - 0.5 * self.volatility **
2) * dt + self.volatility * np.sqrt(dt) *
ran) + (np.exp(self.mu + self.delt *
sn2[t]) - 1) * poi)
self.instrument_values = paths

```

Поскольку мы имеем дело с другой моделью, объект `dx.market_environment` должен включать другой набор элементов. Помимо элементов, передаваемых общему классу моделирования (см. табл. 18.1), нам понадобятся три новых параметра, описывающих прыжковую компоненту с логнормальным распределением: `lambda`, `mu` и `delta` (табл. 18.2).

**Таблица 18.2.** Дополнительные элементы рыночной среды для класса `dx.jump_diffusion`

Элемент	Тип	Обязательный	Описание
<code>lambda</code>	Константа	Да	Интенсивность скачков
<code>mu</code>	Константа	Да	Ожидаемый размер скачка
<code>delta</code>	Константа	Да	Стандартное отклонение размера скачка

Для расчета траекторий этому классу требуются дополнительные случайные числа, представляющие прыжковую компоненту модели. Они генерируются в двух местах (см. комментарии в коде метода `generate_paths()`). О генерировании случайных чисел с распределением Пуассона рассказывалось в главе 12.

## Пример использования

В следующем интерактивном сеансе показано применение класса `dx.jump_diffusion`. За основу здесь взят объект рыночной среды `dx.market_environment`, созданный для модели геометрического броуновского движения.

```

In [47]: me_jd = market_environment('me_jd', dt.datetime(2020, 1, 1))

In [48]: me_jd.add_constant('lambda', 0.3) ❶
          me_jd.add_constant('mu', -0.75) ❶
          me_jd.add_constant('delta', 0.1) ❶

```

```
In [49]: me_jd.add_environment(me_gbm) ❷
```

```
In [50]: from jump_diffusion import jump_diffusion
```

```
In [51]: jd = jump_diffusion('jd', me_jd)
```

```
In [52]: %time paths_3 = jd.get_instrument_values() ❸  
CPU times: user 28.6 ms, sys: 4.37 ms, total: 33 ms  
Wall time: 49.4 ms
```

```
In [53]: jd.update(lamb=0.9) ❹
```

```
In [54]: %time paths_4 = jd.get_instrument_values() ❺  
CPU times: user 29.7 ms, sys: 3.58 ms, total: 33.3 ms  
Wall time: 66.7 ms
```

- ❶ Три дополнительных параметра для объекта `dx.jump_diffusion`. Они специфичны для конкретного класса моделирования.
- ❷ Добавление базовых параметров среды.
- ❸ Моделирование траекторий с заданными параметрами.
- ❹ Увеличение интенсивности скачков.
- ❺ Моделирование траекторий с обновленными параметрами.

На рис. 18.2 сравниваются траектории для моделей с низкой и высокой интенсивностью (вероятностью скачков). В модели с низкой интенсивностью наблюдается всего несколько скачков, тогда как в модели с высокой интенсивности их намного больше.

```
In [55]: plt.figure(figsize=(10, 6))  
p1 = plt.plot(gbm.time_grid, paths_3[:, :10], 'b')  
p2 = plt.plot(gbm.time_grid, paths_4[:, :10], 'r-.')  
l1 = plt.legend([p1[0], p2[0]],  
                ['Низкая интенсивность',  
                 'Высокая интенсивность'], loc=3)  
plt.gca().add_artist(l1)  
plt.xticks(rotation=30);
```

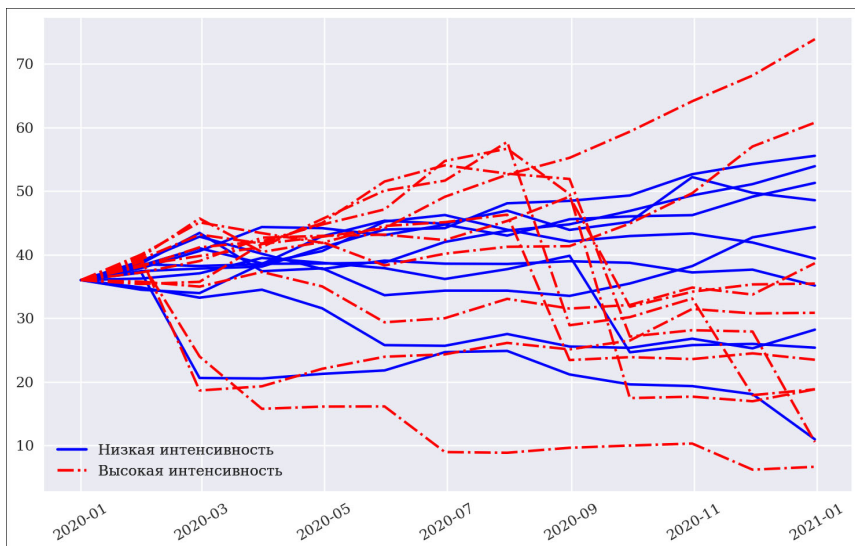


Рис. 18.2. Смоделированные траектории прыжковой диффузии

## Диффузия по закону квадратного корня

Третий тип рассматриваемых нами случайных процессов — диффузия по закону квадратного корня, предложенная Коксом, Ингерсоллом и Россом [2] для моделирования краткосрочных ставок. Соответствующее стохастическое дифференциальное уравнение имеет следующий вид (см. также уравнение 12.4).

**Уравнение 18.5.** Стохастическое дифференциальное уравнение для диффузии по закону квадратного корня

$$dx_t = k(\theta - x_t)dt + \sigma\sqrt{x_t}dZ_t.$$

Мы применим схему дискретизации, представленную уравнением 18.6 (см. уравнение 12.5, а также уравнение 12.6, описывающее альтернативную, более точную схему).

**Уравнение 18.6.** Метод Эйлера для дискретизации модели диффузии по закону квадратного корня

$$\begin{aligned}\tilde{x}_{t_{m+1}} &= \tilde{x}_{t_m} + k(\theta - \tilde{x}_S^+)(t_{m+1} - t_m) + \sigma\sqrt{\tilde{x}_S^+}\sqrt{t_{m+1} - t_m}Z_t, \\ x_{t_{m+1}} &= \tilde{x}_{t_{m+1}}^+.\end{aligned}$$

## Класс моделирования

Ниже показан код класса `dx.square_root_diffusion`, последнего в этой главе. Помимо, опять-таки, другой модели и схемы дискретизации, класс не содержит ничего нового по сравнению с двумя другими классами моделирования.

```
#
# Пакет DX
#
# Класс моделирования -- диффузия по закону квадратного корня
#
# square_root_diffusion.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class

class square_root_diffusion(simulation_class):
    ''' Класс, генерирующий траектории на основе
        модели диффузии по закону квадратного корня

    Атрибуты
    =====
    name: строка
           Имя объекта
    mkt_env: экземпляр класса market_environment
           Рыночная среда для моделирования
    corr: булево значение
           True, если есть корреляция с другим объектом

    Методы
    =====
    update:
           Обновление параметров
    generate_paths:
           Возвращает траектории, рассчитанные по методу
            Монте-Карло для заданной рыночной среды
    ...
```



```

def __init__(self, name, mar_env, corr=False):
    super(square_root_diffusion, self).__init__(name, mar_env,
                                                corr)

    # Требуется дополнительные параметры
    self.kappa = mar_env.get_constant('kappa')
    self.theta = mar_env.get_constant('theta')

def update(self, initial_value=None, volatility=None,
           kappa=None, theta=None, final_date=None):
    if initial_value is not None:
        self.initial_value = initial_value
    if volatility is not None:
        self.volatility = volatility
    if kappa is not None:
        self.kappa = kappa
    if theta is not None:
        self.theta = theta
    if final_date is not None:
        self.final_date = final_date
    self.instrument_values = None

def generate_paths(self, fixed_seed=True, day_count=365.):
    if self.time_grid is None:
        self.generate_time_grid()
    M = len(self.time_grid)
    I = self.paths
    paths = np.zeros((M, I))
    paths_ = np.zeros_like(paths)
    paths[0] = self.initial_value
    paths_[0] = self.initial_value
    if self.correlated is False:
        rand = sn_random_numbers((1, M, I),
                                fixed_seed=fixed_seed)
    else:
        rand = self.random_numbers

    for t in range(1, len(self.time_grid)):
        dt = (self.time_grid[t] - self.time_grid[t - 1]).days /
            day_count
        if self.correlated is False:
            ran = rand[t]
        else:
            ran = np.dot(self.cholesky_matrix, rand[:, t, :])
            ran = ran[self.rn_set]

```

```

# Метод Эйлера
paths_[t] = (paths_[t - 1] + self.kappa *
             (self.theta - np.maximum(0,
             paths_[t - 1, :])) * dt +
             np.sqrt(np.maximum(0, paths_[t - 1, :])) *
             self.volatility * np.sqrt(dt) * ran)
paths[t] = np.maximum(0, paths_[t])
self.instrument_values = paths

```

Два дополнительных элемента рыночной среды, специфичных для данного класса, описаны в табл. 18.3.

*Таблица 18.3. Дополнительные элементы рыночной среды для класса `dx.square_root_diffusion`*

Элемент	Тип	Обязательный	Описание
kappa	Константа	Да	Коэффициент возврата к среднему
theta	Константа	Да	Долгосрочное среднее процесса

## Пример использования

Ниже приведен короткий пример использования класса `dx.square_root_diffusion`. Как обычно, сначала нужно создать объект рыночной среды для моделирования волатильности.

```
In [56]: me_srd = market_environment('me_srd',
                                     dt.datetime(2020, 1, 1)) ❶
```

```
In [57]: me_srd.add_constant('initial_value', .25)
me_srd.add_constant('volatility', 0.05)
me_srd.add_constant('final_date', dt.datetime(2020, 12, 31))
me_srd.add_constant('currency', 'EUR')
me_srd.add_constant('frequency', 'W')
me_srd.add_constant('paths', 10000)
```

```
In [58]: me_srd.add_constant('kappa', 4.0)
me_srd.add_constant('theta', 0.2)
```

```
In [59]: me_srd.add_curve('discount_curve',
                           constant_short_rate('r', 0.0)) ❷
```

```
In [60]: from square_root_diffusion import square_root_diffusion
```

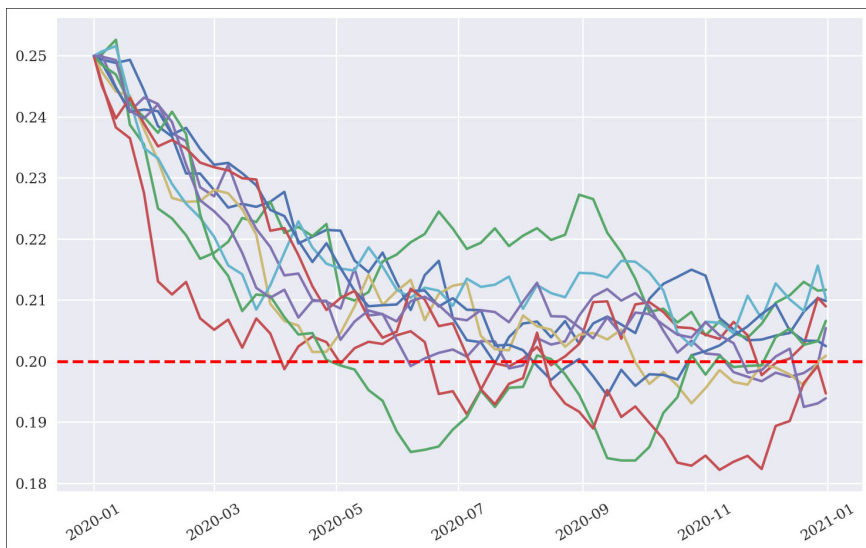
```
In [61]: srd = square_root_diffusion('srd', me_srd) ③
```

```
In [62]: srd_paths = srd.get_instrument_values()[:, :10] ④
```

- ① Дополнительные параметры рыночной среды для объекта `dx.square_root_diffusion`
- ② Объект `discount_curve` требуется по умолчанию, но не используется в моделировании.
- ③ Создание объекта модели.
- ④ Моделирование траекторий и выбор 10 из них.

На рис. 18.3 показано, как выглядит возврат к среднему. Моделируемые траектории стремятся к уровню долгосрочного среднего `theta` (пунктирная линия), равного 0,2.

```
In [63]: plt.figure(figsize=(10, 6))  
         plt.plot(srd.time_grid, srd.get_instrument_values()[:, :10])  
         plt.axhline(me_srd.get_constant('theta'), color='r',  
                     ls='--', lw=2.0)  
         plt.xticks(rotation=30);
```



*Рис. 18.3. Смоделированные траектории диффузии по закону квадратного корня (пунктирной линией обозначен уровень долгосрочного среднего — `theta`)*

## Резюме

В этой главе мы разработали классы, применяемые для моделирования трех случайных процессов: геометрического броуновского движения, прыжковой диффузии и диффузии по закону квадратного корня. Сначала мы написали вспомогательную функцию, генерирующую случайные числа со стандартным нормальным распределением. Затем мы создали общий класс моделирования, на основе которого реализовали три специальных класса и показали примеры их использования.

Чтобы упростить последующий импорт модулей, мы создадим новый оболочечный модуль `dx_simulation.py`.

```
#
# Пакет DX
#
# Функции и классы моделирования
#
# dx_simulation.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd

from dx_frame import *
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion
```

Как и в случае первого оболочечного модуля, `dx_frame.py`, подключение всех необходимых библиотек можно будет выполнять с помощью единственной команды импорта:

```
from dx_simulation import *
```

Поскольку модуль `dx_simulation.py` импортирует все содержимое модуля `dx_frame.py`, тем самым мы подключаем всю имеющуюся на данный момент функциональность. То же самое справедливо для обновленного модуля `__init__.py`, хранящегося в папке `dx`.

```

#
# Пакет DX
#
# Пакетный файл
#
# __init__.py
#
import numpy as np
import pandas as pd
import datetime as dt

# Базовые инструменты
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment

# Моделирование
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion

```

## Дополнительные ресурсы

В главе упоминались следующие источники.

1. Black, Fischer, and Myron Scholes. *The Pricing of Options and Corporate Liabilities* (1973, *Journal of Political Economy*, Vol. 81, No. 3, pp. 637–654).
2. Cox, John, Jonathan Ingersoll, and Stephen Ross. *A Theory of the Term Structure of Interest Rates* (1985, *Econometrica*, Vol. 53, No. 2, pp. 385–407).
3. Glasserman, Paul. *Monte Carlo Methods in Financial Engineering* (2004, Springer).
4. Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).
5. Merton, Robert. *Theory of Rational Option Pricing* (1973, *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141–183).
6. Merton, Robert. *Option Pricing When the Underlying Stock Returns Are Discontinuous* (1976, *Journal of Financial Economics*, Vol. 3, No. 3, pp. 125–144).

## Оценка деривативов

Деривативы — это огромная, сложная проблема.

Джадд Грегг

В течение длительного времени оценкой опционов и других деривативов занимались только аналитики Уолл-стрит — люди с ученой степенью в области точных наук, предполагающих углубленное знание математики. Однако для практического применения моделей, описываемых численными методами наподобие метода Монте-Карло, требуется намного меньше знаний, чем для разработки самих моделей.

Это особенно справедливо в отношении *европейских опционов*, которые могут погашаться только в конкретную дату, и в меньшей степени — в отношении *американских опционов*, которые могут погашаться в любой момент на протяжении определенного периода. В этой главе мы рассмотрим *метод наименьших квадратов Монте-Карло* (Least-Squares Monte Carlo — LSM) — эталонный алгоритм оценки американских опционов, основанный на методе Монте-Карло.

Эта глава организована примерно так же, как и предыдущая. Сначала мы познакомимся с общим классом оценки деривативов, а затем рассмотрим два специализированных класса — для европейского и американского опционов. Общий класс содержит методы для численной оценки наиболее важных греческих коэффициентов опциона: *дельты* и *веги*. Таким образом, эти классы важны не только для задач прогнозирования стоимости ценных бумаг, но и для *управления рисками*.

В главе рассматриваются следующие темы.

### *Общий класс оценки деривативов*

В этом разделе описан общий класс оценки деривативов, от которого наследуются все остальные классы.

### *Европейский опцион*

В этом разделе рассматривается класс оценки европейских опционов.

### *Американский опцион*

В этом разделе рассматривается класс оценки американских опционов.

## Общий класс оценки деривативов

Как и в случае общего класса моделирования, конструктору класса оценки необходимо передать несколько параметров (в данном случае четыре).

`name`

Имя объекта модели.

`underlying`

Экземпляр класса моделирования, представляющий базовый актив.

`mag_env`

Экземпляр класса `dx.market_environment`

`payoff_func`

Строка, представляющая функцию выплат по опциону.

Общий класс оценки включает три метода.

`update()`

Обновляет выбранные параметры (атрибуты).

`delta()`

Вычисляет дельту опциона.

`vega()`

Вычисляет вегу опциона.

Поскольку из предыдущей главы вы уже знаете о том, как устроен пакет DX, вы легко разберетесь в структуре общего класса оценки. Везде, где необходимо, предоставлены поясняющие комментарии. Сначала приводится код класса целиком, а затем мы разберем ключевые нюансы.

```
#
# Пакет DX
#
# Оценка деривативов -- базовый класс
#
# valuation_class.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#

class valuation_class(object):
    ''' Базовый класс однофакторной оценки.
```

## **Атрибуты**

=====

**name:** строка

Имя объекта

**underlying:** экземпляр класса моделирования

Объект, моделирующий одиночный фактор риска

**mar\_env:** экземпляр класса `market_environment`

Рыночная среда для оценки дериватива

**payoff\_func:** строка

Выплаты по деривативу в синтаксисе Python.

Пример: `'np.maximum(maturity_value - 100, 0)'`,  
где `maturity_value` - NumPy-вектор соответствующих  
стоимостей базового актива.

Пример: `'np.maximum(instrument_values - 100, 0)'`,  
где `instrument_values` - NumPy-матрица стоимостей  
базового актива за все время/интервал моделирования.

## **Методы**

=====

**update:**

Обновление выбранных параметров

**delta:**

Возвращает дельту дериватива

**vega:**

Возвращает вегу дериватива

'''

```
def __init__(self, name, underlying, mar_env, payoff_func=''):
    self.name = name
    self.pricing_date = mar_env.pricing_date
    try:
        # Необязательный параметр
        self.strike = mar_env.get_constant('strike')
    except:
        pass
    self.maturity = mar_env.get_constant('maturity')
    self.currency = mar_env.get_constant('currency')
    # Параметры моделирования и кривая дисконтирования
    self.frequency = underlying.frequency
    self.paths = underlying.paths
    self.discount_curve = underlying.discount_curve
    self.payoff_func = payoff_func
```



```

self.underlying = underlying
# Передача параметров pricing_date
# и maturity базовому активу
self.underlying.special_dates.extend([self.pricing_date,
                                      self.maturity])

def update(self, initial_value=None, volatility=None,
           strike=None, maturity=None):
    if initial_value is not None:
        self.underlying.update(initial_value=initial_value)
    if volatility is not None:
        self.underlying.update(volatility=volatility)
    if strike is not None:
        self.strike = strike
    if maturity is not None:
        self.maturity = maturity
        # Добавляем новую дату исполнения, если она
        # отсутствует в сетке time_grid
        if maturity not in self.underlying.time_grid:
            self.underlying.special_dates.append(maturity)
            self.underlying.instrument_values = None

def delta(self, interval=None, accuracy=4):
    if interval is None:
        interval = self.underlying.initial_value / 50.
    # Метод конечных разностей

    # Вычисление значения слева в формуле дельты
    value_left = self.present_value(fixed_seed=True)
    # Стоимость базового актива для значения справа
    initial_del = self.underlying.initial_value + interval
    self.underlying.update(initial_value=initial_del)
    # Вычисление значения справа в формуле дельты
    value_right = self.present_value(fixed_seed=True)
    # Сброс параметра initial_value моделируемого объекта
    self.underlying.update(initial_value=initial_del - interval)
    delta = (value_right - value_left) / interval
    # Поправка на случай возможных ошибок аппроксимации
    if delta < -1.0:
        return -1.0
    elif delta > 1.0:
        return 1.0
    else:

```

```

        return round(delta, accuracy)

def vega(self, interval=0.01, accuracy=4):
    if interval < self.underlying.volatility / 50.:
        interval = self.underlying.volatility / 50.
    # Метод конечных разностей

    # Вычисление значения слева в формуле веги
    value_left = self.present_value(fixed_seed=True)
    # Волатильность для значения справа
    vola_del = self.underlying.volatility + interval
    # Обновление моделируемого объекта
    self.underlying.update(volatility=vol_a_del)
    # Вычисление значения справа в формуле веги
    value_right = self.present_value(fixed_seed=True)
    # Сброс параметра volatility моделируемого объекта
    self.underlying.update(volatility=vol_a_del - interval)
    vega = (value_right - value_left) / interval
    return round(vega, accuracy)

```

Одна из задач, решаемых с помощью класса `dx.valuation_class`, — оценка “греков” опциона. Этот вопрос следует рассмотреть подробнее. Предположим, что дисконтированная стоимость опциона описывается непрерывно дифференцируемой функцией  $V(S_0, \sigma_0)$ . Тогда *дельта* опциона определяется как первая частная производная этой функции по текущей стоимости базового актива  $S_0$ , или  $\Delta = \frac{\partial V(\cdot)}{\partial S_0}$ .

Предположим далее, что для стоимости опциона имеется численная оценка  $\bar{V}(S_0, \sigma_0)$ , полученная по методу Монте-Карло (см. главу 12). Тогда численная аппроксимация дельты опциона описывается уравнением 19.1<sup>1</sup>. Для решения этой задачи в класс оценки добавлен метод `delta()`. В нем предполагается существование метода `present_value()`, который возвращает оценку по методу Монте-Карло для заданного набора параметров.

<sup>1</sup> Детали численной оценки “греков” по методу Монте-Карло приведены у Глассермана [2]. В коде реализуется только схема с *правыми разностями*, так как в этом случае требуется выполнить всего *одно* дополнительное моделирование опциона с последующей переоценкой. Например, в случае схемы с *центральными разностями* потребуются выполнить сразу *две* дополнительные переоценки опциона, что приведет к увеличению вычислительной нагрузки.

### Уравнение 19.1. Численная аппроксимация дельты опциона

$$\bar{\Delta} = \frac{\bar{V}(S_0 + \Delta S, \sigma_0) - \bar{V}(S_0, \sigma_0)}{\Delta S}, \Delta S > 0.$$

Аналогичным образом рассчитывается *vega* опциона. Этот коэффициент определяется как первая частная производная дисконтированной стоимости опциона по его текущей (мгновенной) волатильности:  $V = \frac{\partial V(\cdot)}{\partial \sigma_0}$ . Если имеется оценка стоимости опциона, полученная по методу Монте-Карло, то численная аппроксимация вего опциона описывается уравнением 19.2. Для решения этой задачи в класс `dx.valuation_class` добавлен метод `vega()`.

### Уравнение 19.2. Численная аппроксимация вего опциона

$$V = \frac{\bar{V}(S_0, \sigma_0 + \Delta \sigma) - \bar{V}(S_0, \sigma_0)}{\Delta \sigma}, \Delta \sigma > 0.$$

Заметьте, что вычисление дельты и вего базируется на существовании либо дифференцируемой функции, либо оценки дисконтированной стоимости опциона, полученной по методу Монте-Карло. Именно по этой причине можно определить методы для численной оценки “греков”, не зная, как конкретно получена оценка стоимости по методу Монте-Карло.

## Европейский опцион

Первый специализированный класс оценки, который мы рассмотрим, — это класс европейских опционов. Рассмотрим следующую упрощенную схему для получения оценки стоимости опциона по методу Монте-Карло.

1. Смоделировать соответствующий базовый фактор риска  $S$  с риск-нейтральной мерой  $I$  раз для получения соответствующего количества смоделированных значений базового актива на дату исполнения опциона  $T$ , т.е.  $\bar{S}_T(i)$ ,  $i \in \{1, 2, \dots, I\}$ .
2. Вычислить премию  $h_T$  опциона на дату исполнения для каждого смоделированного значения базового актива, т.е.  $h_T(\bar{S}_T(i))$ ,  $i \in \{1, 2, \dots, I\}$ .
3. Получить оценку дисконтированной стоимости опциона по методу Монте-Карло:  $\bar{V}_0 \equiv e^{-rT} \frac{1}{I} \sum_{i=1}^I h_T(\bar{S}_T(i))$ .

## Класс оценки

Ниже показан код класса, в котором реализован метод `present_value()` по описанной схеме. Также класс включает метод `generate_payoff()`, который генерирует смоделированные траектории и на их основе вычисляет премию опциона. Это позволяет получить оценку по методу Монте-Карло.

```
#
# Пакет DX
#
# Оценка деривативов -- класс европейского опциона
#
# valuation_mcs_european.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from valuation_class import valuation_class

class valuation_mcs_european(valuation_class):
    ''' Класс однофакторной оценки европейских опционов
        с произвольными выплатами, выполняемой по методу
        Монте-Карло.

        Методы
        =====
        generate_payoff:
            Возвращает выплаты для заданных траекторий
            и функции выплат
        present_value:
            Возвращает дисконтированную стоимость
            (оценка по методу Монте-Карло)
        ...

    def generate_payoff(self, fixed_seed=False):
        '''
        Параметры
        =====
        fixed_seed: булево значение
            Использовать то же самое (фиксированное) затравочное
            значение для оценки
        ...
```

```

try:
    # Необязательный параметр
    strike = self.strike
except:
    pass
paths = self.underlying.get_instrument_values(
    fixed_seed=fixed_seed)
time_grid = self.underlying.time_grid
try:
    time_index = np.where(time_grid == self.maturity)[0]
    time_index = int(time_index)
except:
    print('Maturity date not in time grid of underlying.')
maturity_value = paths[time_index]
# Средняя стоимость по всей траектории
mean_value = np.mean(paths[:time_index], axis=1)
# Максимальная стоимость по всей траектории
max_value = np.amax(paths[:time_index], axis=1)[-1]
# Минимальная стоимость по всей траектории
min_value = np.amin(paths[:time_index], axis=1)[-1]
try:
    payoff = eval(self.payoff_func)
    return payoff
except:
    print('Error evaluating payoff function.')

def present_value(self, accuracy=6, fixed_seed=False,
                  full=False):
    '''
    Параметры
    =====
    accuracy: целочисленное значение
        Количество десятичных знаков в возвращаемом значении
    fixed_seed: булево значение
        Использовать то же самое (фиксированное) затравочное
        значение для оценки
    full: булево значение
        Возвращать также полный одномерный массив
        дисконтированных стоимостей
    '''
    cash_flow = self.generate_payoff(fixed_seed=fixed_seed)
    discount_factor = self.discount_curve.get_discount_factors(
        (self.pricing_date, self.maturity))[0, 1]

```

```

result = discount_factor * np.sum(cash_flow) / len(cash_flow)
if full:
    return round(result, accuracy),
        discount_factor * cash_flow
else:
    return round(result, accuracy)

```

Для расчета выплат по опциону метод `generate_payoff()` определяет ряд специальных объектов.

- **strike.** Страйк-цена опциона.
- **maturity\_value.** Одномерный объект `ndarray`, содержащий смоделированные значения базового актива на дату исполнения опциона.
- **mean\_value.** Среднее значение базового актива по всей траектории за период от текущей даты до даты исполнения.
- **max\_value.** Максимальное значение базового актива по всей траектории.
- **min\_value.** Минимальное значение базового актива по всей траектории.

Последние три параметра позволяют обрабатывать азиатские опционы (в которых цена исполнения вычисляется по средней стоимости за определенный период).



#### Гибкие выплаты

Подход к оценке стоимости европейских опционов является довольно гибким в том смысле, что он разрешает задавать произвольную функцию выплат. Среди прочего это дает возможность моделировать как деривативы с условным исполнением (т.е. опционы), так и с безусловным исполнением (т.е. форварды). Это также позволяет учитывать выплаты по экзотическим опционам, например азиатским.

## Пример использования

Рассмотрим конкретный пример использования класса `dx.valuation_mcs_euroean`. Прежде чем создавать экземпляр класса, необходимо создать моделируемый объект, т.е. базовый актив оцениваемого опциона. Для этого мы воспользуемся классом `dx.geometric_brownian_motion`, который был создан в главе 18.

```

In [64]: me_gbm = market_environment('me_gbm',
                                     dt.datetime(2020, 1, 1))

In [65]: me_gbm.add_constant('initial_value', 36.)
me_gbm.add_constant('volatility', 0.2)
me_gbm.add_constant('final_date', dt.datetime(2020, 12, 31))
me_gbm.add_constant('currency', 'EUR')
me_gbm.add_constant('frequency', 'M')
me_gbm.add_constant('paths', 10000)

In [66]: csr = constant_short_rate('csr', 0.06)

In [67]: me_gbm.add_curve('discount_curve', csr)

In [68]: gbm = geometric_brownian_motion('gbm', me_gbm)

```

Кроме объекта моделирования, нам нужно определить рыночную среду для опциона. Она должна как минимум включать параметры `maturity` и `currency`. Также может предоставляться параметр `strike`.

```

In [69]: me_call = market_environment('me_call', me_gbm.pricing_date)

In [70]: me_call.add_constant('strike', 40.)
me_call.add_constant('maturity', dt.datetime(2020, 12, 31))
me_call.add_constant('currency', 'EUR')

```

Но ключевой элемент — это, конечно же, функция выплат, которая предоставляется в виде строки, содержащей код Python для функции `eval()`. Мы будем моделировать европейский *колл-опцион*. Для такого опциона функция выплат имеет вид  $h_T = \max(S_T - K, 0)$ , где  $S_T$  — стоимость базового актива на дату исполнения, а  $K$  — страйк-цена опциона. Благодаря векторизованным инструментам NumPy это уравнение записывается следующим образом:

```

In [71]: payoff_func = 'np.maximum(maturity_value - strike, 0)'

```

Теперь, когда все необходимые параметры предоставлены, можно создать экземпляр класса `dx.valuation_mcs_european`. Все интересующие нас значения можно вычислить с помощью единственного вызова.

```

In [72]: from valuation_mcs_european import valuation_mcs_european

In [73]: eur_call = valuation_mcs_european('eur_call',
                                     underlying=gbm,
                                     mar_env=me_call,
                                     payoff_func=payoff_func)

```

```
In [74]: %time eur_call.present_value() ❶
CPU times: user 14.8 ms, sys: 4.06 ms, total: 18.9 ms
Wall time: 43.5 ms
Out[74]: 2.146828
```

```
In [75]: %time eur_call.delta() ❷
CPU times: user 12.4 ms, sys: 2.68 ms, total: 15.1 ms
Wall time: 40.1 ms
Out[75]: 0.5155
```

```
In [76]: %time eur_call.vega() ❸
CPU times: user 21 ms, sys: 2.72 ms, total: 23.7 ms
Wall time: 89.9 ms
Out[76]: 14.301
```

- ❶ Оценка дисконтированной стоимости европейского колл-опциона.
- ❷ Числовая оценка дельты опциона. У колл-опционов дельта может быть только положительной.
- ❸ Числовая оценка веги опциона. Вега будет положительной как у колл-опционов, так и у пут-опционов.

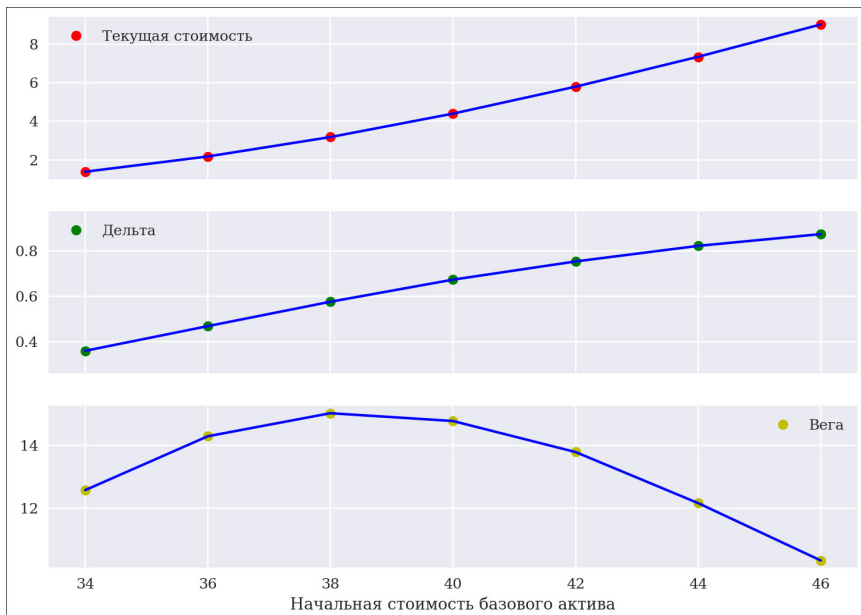
Имея объект оценки, можно выполнить более сложный анализ дисконтированной стоимости и “греков” опциона. В следующем коде вычисляется текущая стоимость вместе с дельтой и вегой опциона для начальных стоимостей базового актива в диапазоне от 34 до 46 евро. Результаты моделирования представлены на рис. 19.1.

```
In [77]: %%time
s_list = np.arange(34., 46.1, 2.)
p_list = []; d_list = []; v_list = []
for s in s_list:
    eur_call.update(initial_value=s)
    p_list.append(eur_call.present_value(fixed_seed=True))
    d_list.append(eur_call.delta())
    v_list.append(eur_call.vega())
CPU times: user 374 ms, sys: 8.82 ms, total: 383 ms
Wall time: 609 ms
```

```
In [78]: from plot_option_stats import plot_option_stats
```

```
In [79]: plot_option_stats(s_list, p_list, d_list, v_list)
```





**Рис. 19.1.** *Оценки текущей (дисконтированной) стоимости, дельты и веги европейского колл-опциона*

Для визуализации применяется вспомогательная функция `plot_option_stats()`.

```
#
# Пакет DX
#
# Оценка деривативов -- вывод статистики опциона
#
# plot_option_stats.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import matplotlib.pyplot as plt

def plot_option_stats(s_list, p_list, d_list, v_list):
    ''' Строит графики текущей стоимости, дельты и веги опциона
        для набора начальных стоимостей базового актива.
```

### *Параметры*

=====

*s\_list: массив или список*

*Начальные стоимости базового актива*

*p\_list: массив или список*

*Дисконтированные стоимости*

*d\_list: массив или список*

*Значения дельты*

*v\_list: массив или список*

*Значения веги*

'''

```
plt.figure(figsize=(10, 7))
sub1 = plt.subplot(311)
plt.plot(s_list, p_list, 'ro', label='Текущая стоимость')
plt.plot(s_list, p_list, 'b')
plt.legend(loc=0)
plt.setp(sub1.get_xticklabels(), visible=False)
sub2 = plt.subplot(312)
plt.plot(s_list, d_list, 'go', label='Дельта')
plt.plot(s_list, d_list, 'b')
plt.legend(loc=0)
plt.ylim(min(d_list) - 0.1, max(d_list) + 0.1)
plt.setp(sub2.get_xticklabels(), visible=False)
sub3 = plt.subplot(313)
plt.plot(s_list, v_list, 'yo', label='Bera')
plt.plot(s_list, v_list, 'b')
plt.xlabel('Начальная стоимость базового актива')
plt.legend(loc=0)
```

Этот пример наглядно показывает, что, несмотря на сложные математические вычисления, подход, реализованный в пакете DX, сопоставим с наличием замкнутой формулы оценки опционов. Причем данный подход применим не только к стандартным опционам, которые мы рассматривали до сих пор, но и к экзотическим опционам с более сложными выплатами.

Рассмотрим следующую схему выплат, в которой сочетаются модели стандартного и азиатского опционов. Алгоритм работы остается тем же и фактически не зависит от типа выплат. На рис. 19.2 видно, что в данном случае дельта достигает значения 1 тогда, когда начальная стоимость базового актива достигает страйк-цены 40. С этого момента любое дальнейшее увеличение начальной стоимости приводит к равнозначному повышению стоимости опциона.

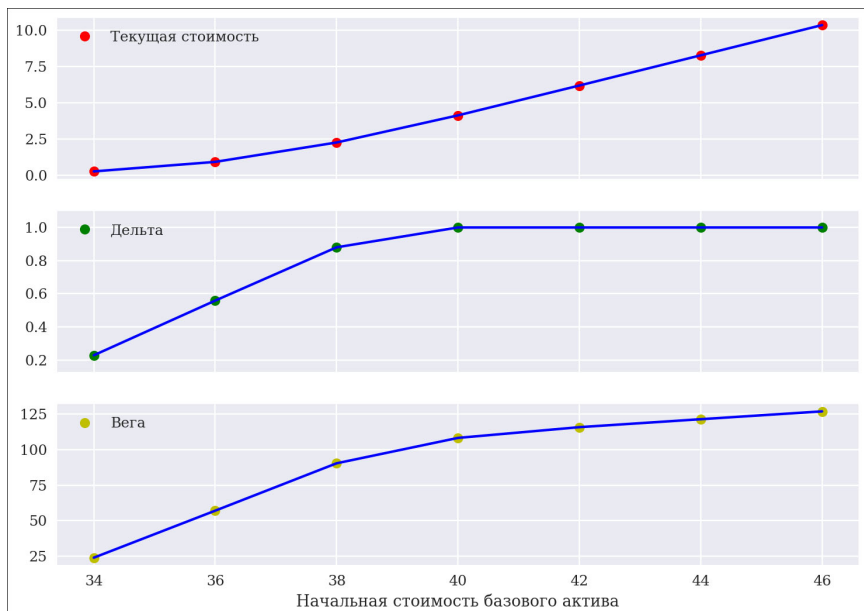


Рис. 19.2. Оценки текущей (дисконтированной) стоимости, дельты и веги азиатского опциона

```
In [80]: payoff_func = 'np.maximum(0.33 * '
        payoff_func += '(maturity_value + max_value) - 40, 0)'
```

❶

```
In [81]: eur_as_call = valuation_mcs_european('eur_as_call',
        underlying=gbm,
        mar_env=me_call,
        payoff_func=payoff_func)
```

```
In [82]: %%time
        s_list = np.arange(34., 46.1, 2.)
        p_list = []; d_list = []; v_list = []
        for s in s_list:
            eur_as_call.update(s)
            p_list.append(eur_as_call.present_value(
                fixed_seed=True))
            d_list.append(eur_as_call.delta())
            v_list.append(eur_as_call.vega())
        CPU times: user 319 ms, sys: 14.2 ms, total: 333 ms
        Wall time: 488 ms
```

```
In [83]: plot_option_stats(s_list, p_list, d_list, v_list)
```

- ❶ Премия опциона зависит как от прогнозируемой цены исполнения, так и от максимальной цены в моделируемой траектории.

## Американский опцион

По сравнению с европейскими опционами оценивать американские или бермудские опционы сложнее<sup>2</sup>. Прежде чем переходить к созданию класса оценки, необходимо познакомиться с теоретическими основами.

## Метод наименьших квадратов Монте-Карло

Несмотря на то что простой численный метод совместной оценки европейских и американских опционов был предложен еще в биномиальной модели Кокса, Росса и Рубинштейна [1], проблема оценки американского опциона методом Монте-Карло была удовлетворительно решена только после публикации работы Лонгстаффа — Шварца [5]. Основная трудность заключается в том, что метод Монте-Карло по своей сути — алгоритмом прямого прохода, тогда как оценка американских опционов реализуется в обратном направлении: продленная стоимость опциона оценивается от даты исполнения *назад* к текущему моменту времени.

Идея модели Лонгстаффа — Шварца сводится к использованию обыкновенной регрессии по методу наименьших квадратов для оценки продленной стоимости на основе перекрестных данных по всем доступным смоделированным значениям<sup>3</sup>. Для каждой траектории учитывается следующее:

- смоделированная стоимость базового актива (или активов);
- внутренняя стоимость опциона;
- действительная продленная стоимость по конкретной траектории.

---

<sup>2</sup> Американский опцион может быть исполнен в любой момент времени в течение оговоренного периода (как минимум, в течение торгового дня). Бермудский опцион предполагает наличие нескольких дискретных дат исполнения. В программах моделирования американский опцион аппроксимируется бермудским опционом, в котором количество дат исполнения стремится к бесконечности.

<sup>3</sup> Именно поэтому такой алгоритм сокращенно называют методом наименьших квадратов Монте-Карло (Least-Squares Monte Carlo — LSM).

В дискретном времени расчет стоимости бермудского опциона (а в непрерывном времени — американского опциона)<sup>4</sup> сводится к задаче *оптимального момента остановки* (марковского момента), описываемой уравнением 19.3 для конечного набора временных точек  $0 < t_1 < t_2 < \dots < T$ .

**Уравнение 19.3. Задача оптимального момента остановки в дискретном времени для бермудского опциона**

$$V_0 = \sup_{\tau \in \{0, t_1, t_2, \dots, T\}} e^{-r\tau} \mathbb{E}_0^Q \left( h_\tau (S_\tau) \right).$$

Уравнение 19.4 применяется для вычисления продленной стоимости американского опциона в диапазоне дат  $0 < t_m < T$ . Она определяется как риск-нейтральная ожидаемая стоимость в момент времени  $t_m$ , выраженная через мартингальную меру стоимости американского опциона  $V_{t_{m+1}}$  в следующую дату.

**Уравнение 19.4. Продленная стоимость американского опциона**

$$C_{t_m}(s) = e^{-r(t_{m+1}-t_m)} \mathbb{E}_{t_m}^Q \left( V_{t_{m+1}}(S_{t_{m+1}}) \mid S_{t_m} = s \right).$$

Стоимость американского опциона  $V_{t_m}$  в момент времени  $t_m$  можно рассчитать с помощью уравнения 19.5 как максимальное из двух значений: премия при немедленном исполнении (внутренняя стоимость) и ожидаемая премия в отсутствие исполнения (продленная стоимость).

**Уравнение 19.5. Стоимость американского опциона на заданную дату**

$$V_{t_m} = \max \left( h_{t_m}(s), C_{t_m}(s) \right).$$

Внутренняя стоимость опциона, используемая в уравнении 19.5, вычисляется очень легко. Труднее определить продленную стоимость. В алгоритме Лонгстаффа — Шварца [5] это значение аппроксимируется регрессионным путем, как показано в уравнении 19.6. Здесь  $i$  обозначает текущую моделируемую траекторию,  $D$  — количество базисных функций, используемых в регрессионном анализе,  $\alpha^*$  — оптимальные параметры регрессии, а  $b_d$  — регрессионная функция с номером  $d$ .

<sup>4</sup> У Колера [4] приведен полный обзор теории оценки американских опционов в целом и регрессионных методов в частности.

### Уравнение 19.6. Регрессионная аппроксимация продленной стоимости

$$\bar{C}_{t_m, i} = \sum_{d=1}^D \alpha_{d, t_m}^* b_d(S_{t_m, i}).$$

Оптимальные регрессионные параметры могут быть получены путем решения регрессионного уравнения 19.7 по методу наименьших квадратов. Здесь  $Y_{t_m, i} \equiv e^{-r(t_{m+1}-t_m)} V_{t_{m+1}, i}$  — фактическая (не регрессионная) продленная стоимость на дату  $t_m$  для  $i$ -й траектории.

### Уравнение 19.7. Регрессия по методу наименьших квадратов

$$\min_{\alpha_{1, t_m}, \dots, \alpha_{D, t_m}} - \sum_{i=1}^I \left( Y_{t_m, i} - \sum_{d=1}^D \alpha_{d, t_m} b_d(S_{t_m, i}) \right)^2.$$

На этом мы заканчиваем обзор математического аппарата, применяемого для оценки американских опционов по методу Монте-Карло.

## Класс оценки

Ниже приведен код класса для оценки американских опционов. В нем особого внимания заслуживает один из этапов реализации алгоритма наименьших квадратов в методе `present_value()`: *этап принятия оптимального решения*. Важно то, что на основе этого решения программа выбирает либо внутреннюю стоимость, либо *фактическую* продленную стоимость, но не оценочную продленную стоимость<sup>5</sup>.

```
#
# Пакет DX
#
# Оценка деривативов -- класс американского опциона
#
# valuation_mcs_american.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np

from valuation_class import valuation_class
```

<sup>5</sup> См. главу 6 книги Хилпиша [3].

```

class valuation_mcs_american(valuation_class):
    ''' Класс однофакторной оценки американских опционов
        с произвольными выплатами, выполняемой по методу
        Монте-Карло.

    Методы
    =====
    generate_payoff:
        Возвращает выплаты для заданных траекторий
        и функции выплат
    present_value:
        Возвращает дисконтированную стоимость (оценка
        по методу наименьших квадратов Монте-Карло)
        согласно модели Лонгстаффа – Шварца [5]
    ...

def generate_payoff(self, fixed_seed=False):
    '''
    Параметры
    =====
    fixed_seed: булево значение
        Использовать то же самое (фиксированное) заправочное
        значение для оценки
    ...
    try:
        # Необязательный параметр
        strike = self.strike
    except:
        pass
    paths = self.underlying.get_instrument_values(
        fixed_seed=fixed_seed)
    time_grid = self.underlying.time_grid
    time_index_start = int(np.where(time_grid ==
        self.pricing_date)[0])
    time_index_end = int(np.where(time_grid == self.maturity)[0])
    instrument_values = paths[time_index_start:time_index_end + 1]
    payoff = eval(self.payoff_func)
    return instrument_values, payoff, time_index_start,
        time_index_end

def present_value(self, accuracy=6, fixed_seed=False, bf=5,
        full=False):

```

```

'''
Параметры
=====
accuracy: целочисленное значение
    Количество десятичных знаков в возвращаемом значении
fixed_seed: булево значение
    Использовать то же самое (фиксированное) заправочное
    значение для оценки
bf: целочисленное значение
    Количество базисных функций для регрессии
full: булево значение
    Возвращать также полный одномерный массив
    дисконтированных стоимостей
'''
instrument_values, inner_values, time_index_start, \
    time_index_end = self.generate_payoff(
        fixed_seed=fixed_seed)
time_list = self.underlying.time_grid[
    time_index_start:time_index_end + 1]
discount_factors = self.discount_curve.get_discount_factors(
    time_list, dtobjects=True)
V = inner_values[-1]
for t in range(len(time_list) - 2, 0, -1):
    # Получение соответствующего коэффициента дисконтирования
    # для заданного временного интервала
    df = discount_factors[t, 1] / discount_factors[t + 1, 1]
    # Этап регрессии
    rg = np.polyfit(instrument_values[t], V * df, bf)
    # Вычисление продленных стоимостей для траектории
    C = np.polyval(rg, instrument_values[t])
    # Этап принятия оптимального решения:
    # если условие выполняется (внутренняя стоимость больше
    # регрессионной продленной стоимости), то учитывается
    # внутренняя стоимость; в противном случае учитывается
    # фактическая продленная стоимость
    V = np.where(inner_values[t] > C, inner_values[t], V * df)
df = discount_factors[0, 1] / discount_factors[1, 1]
result = df * np.sum(V) / len(V)
if full:
    return round(result, accuracy), df * V
else:
    return round(result, accuracy)

```



## Пример использования

Проиллюстрируем применение класса `dx.valuation_mcs_american` на конкретном примере, в котором мы попытаемся воспроизвести все оценки стоимости американского опциона, приведенные в работе Лонгстаффа и Шварца [5]. Базовый актив, как и прежде, представляет собой объект `dx.geometric_brownian_motion`. Сначала выполняется параметризация модели.

```
In [84]: me_gbm = market_environment('me_gbm',  
                                     dt.datetime(2020, 1, 1))  
  
In [85]: me_gbm.add_constant('initial_value', 36.)  
me_gbm.add_constant('volatility', 0.2)  
me_gbm.add_constant('final_date', dt.datetime(2021, 12, 31))  
me_gbm.add_constant('currency', 'EUR')  
me_gbm.add_constant('frequency', 'W')  
me_gbm.add_constant('paths', 50000)  
  
In [86]: csr = constant_short_rate('csr', 0.06)  
  
In [87]: me_gbm.add_curve('discount_curve', csr)  
  
In [88]: gbm = geometric_brownian_motion('gbm', me_gbm)  
  
In [89]: payoff_func = 'np.maximum(strike - instrument_values, 0)'  
  
In [90]: me_am_put = market_environment('me_am_put',  
                                         dt.datetime(2020, 1, 1))  
  
In [91]: me_am_put.add_constant('maturity',  
                                dt.datetime(2020, 12, 31))  
me_am_put.add_constant('strike', 40.)  
me_am_put.add_constant('currency', 'EUR')
```

Следующий шаг — создание объекта оценки с заданными параметрами и выполнение самой процедуры оценки. Оценка американского пут-опциона может отнимать намного больше времени, чем в случае европейского опциона. Это связано не только с увеличением количества траекторий и временных интервалов, но и с более сложным вычислительным алгоритмом, требующим выполнения обратных проходов и регрессионных расчетов на каждом из них. Оценка стоимости, полученная для первого опциона, близка к значению, приведенному в оригинальной статье (4,478).

```
In [92]: from valuation_mcs_american import valuation_mcs_american

In [93]: am_put = valuation_mcs_american('am_put', underlying=gbm,
                                         mar_env=me_am_put,
                                         payoff_func=payoff_func)

In [94]: %time am_put.present_value(fixed_seed=True, bf=5)
CPU times: user 1.57 s, sys: 219 ms, total: 1.79 s
Wall time: 2.01 s

Out[94]: 4.472834
```

Суть метода наименьших квадратов Монте-Карло такова, что он позволяет получить *нижний предел* математически правильного значения стоимости американского опциона<sup>6</sup>. Поэтому можно ожидать, что числовая оценка окажется меньше истинного значения в любом реалистичном сценарии. Альтернативный алгоритм двойных оценок позволяет получить *верхний предел*<sup>7</sup>. Совместное использование двух разных методов дает возможность определить интервал, в котором находится истинная стоимость американского опциона.

Как уже было сказано, нашей основной задачей будет повторение всех оценок стоимости американского опциона, приведенных в оригинальной статье Лонгстаффа и Шварца. Для этого нам нужно использовать объект оценки во вложенных циклах. В самом глубоком вложенном цикле объект оценки должен обновляться в соответствии с текущей параметризацией.

```
In [95]: %%time
ls_table = []
for initial_value in (36., 38., 40., 42., 44.):
    for volatility in (0.2, 0.4):
        for maturity in (dt.datetime(2020, 12, 31),
                          dt.datetime(2021, 12, 31)):
            am_put.update(initial_value=initial_value,
                           volatility=volatility,
                           maturity=maturity)
            ls_table.append([initial_value,
                              volatility,
                              maturity,
                              am_put.present_value(bf=5)])
```

---

<sup>6</sup> Основная причина заключается в том, что “оптимальная” политика исполнения, основанная на регрессионной оценке продленной стоимости, в действительности оказывается “субоптимальной”.

<sup>7</sup> Об алгоритме двойных оценок и его реализации на Python рассказано у Хилпиша [3].

```
CPU times: user 41.1 s, sys: 2.46 s, total: 43.5 s
Wall time: 1min 30s
```

```
In [96]: print('S0 | Vola | T | Value')
print(22 * '-')
for r in ls_table:
    print('%d | %3.1f | %d | %5.3f' %
          (r[0], r[1], r[2].year - 2019, r[3]))
```

S0	Vola	T	Value
36	0.2	1	4.447
36	0.2	2	4.773
36	0.4	1	7.006
36	0.4	2	8.377
38	0.2	1	3.213
38	0.2	2	3.645
38	0.4	1	6.069
38	0.4	2	7.539
40	0.2	1	2.269
40	0.2	2	2.781
40	0.4	1	5.211
40	0.4	2	6.756
42	0.2	1	1.556
42	0.2	2	2.102
42	0.4	1	4.466
42	0.4	2	6.049
44	0.2	1	1.059
44	0.2	2	1.617
44	0.4	1	3.852
44	0.4	2	5.490

Полученные результаты представляют собой упрощенную версию таблицы 1 оригинальной статьи Лонгстаффа и Шварца. В целом наши значения достаточно близки к тем, которые были приведены в статье, даже несмотря на несколько иную параметризацию модели (например, у авторов статьи в два раза большее количество траекторий).

В завершение стоит отметить, что оценка “греков” американского опциона формально выполняется так же, как и в случае европейского опциона. В этом заключается основное преимущество реализованного нами подхода перед альтернативными подходами (такими, как биномиальная модель).

```
In [97]: am_put.update(initial_value=36.)
am_put.delta()
Out[97]: -0.4631
```

```
In [98]: am_put.vega()  
Out[98]: 18.0961
```



### Метод наименьших квадратов Монте-Карло

Метод наименьших квадратов Монте-Карло, разработанный Лонгстаффом и Шварцем [5] — очень эффективный численный метод оценки не только стандартных европейских опционов, но и более сложных американских и бермудских опционов. В нем оптимальная стратегия исполнения опциона аппроксимируется с помощью регрессионного метода наименьших квадратов. А поскольку этот метод позволяет легко обрабатывать многомерные данные, его можно успешно применять в том числе для оценки деривативов.

## Резюме

Эта глава была посвящена численной оценке европейских и американских опционов по методу Монте-Карло. Мы разработали общий класс оценки `dx.valuation_class`, содержащий, в частности, методы вычисления наиболее важных “греков” (дельты и веги) опционов обоих типов независимо от объекта моделирования (т.е. фактора риска или стохастического процесса), применяемого для оценки.

На основе общего класса оценки были созданы два специализированных класса: `dx.valuation_mcs_european` и `dx.valuation_mcs_american`. Класс оценки европейских опционов представляет собой достаточно простую реализацию риск-нейтрального подхода, описанного в главе 17. Класс оценки американских опционов требует выполнения регрессионного анализа, алгоритм которого известен как метод наименьших квадратов Монте-Карло. Это связано с тем, что для получения оценки необходимо определить стратегию оптимального исполнения опциона. Такой подход заметно сложнее не только теоретически, но и с точки зрения программной реализации. Тем не менее код соответствующего метода `present_value()` вышел довольно компактным.

Подход к оценке опционов, реализованный в пакете DX, оказывается достаточно удобным. Мы можем без особых усилий оценить относительно большой класс опционов со следующими характеристиками:

- единственный фактор риска;
- европейский или американский тип исполнения;
- произвольные премиальные выплаты.

Помимо этого мы можем рассчитать наиболее важные “греки” опционов рассматриваемого типа. Чтобы упростить последующий импорт всех необходимых модулей, мы снова создадим оболочечный модуль — *dx\_valuation.py*.

```
#
# Пакет DX
#
# Классы оценки
#
# dx_valuation.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd

from dx_simulation import *
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american
```

Соответствующим образом обновляется и файл *\_\_init\_\_.py*, находящийся в папке *dx*.

```
#
# Пакет DX
#
# Пакетный файл
#
# __init__.py
#
import numpy as np
import pandas as pd
import datetime as dt

# Базовые инструменты
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
from plot_option_stats import plot_option_stats

# Моделирование
from sn_random_numbers import sn_random_numbers
```

```
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion
```

```
# Оценка деривативов
```

```
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american
```

## Дополнительные ресурсы

В главе упоминались следующие источники.

1. Cox, John, Stephen Ross, and Mark Rubinstein. *Option Pricing: A Simplified Approach* (1979, *Journal of Financial Economics*, Vol. 7, No. 3, pp. 229–263).
2. Glasserman, Paul. *Monte Carlo Methods in Financial Engineering* (2004, Springer).
3. Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).
4. Kohler, Michael. *A Review on Regression-Based Monte Carlo Methods for Pricing American Options*. In Luc Devroye et al. (eds.): *Recent Developments in Applied Probability and Statistics* (2010, pp. 37–58, Physica-Verlag).
5. Longstaff, Francis, and Eduardo Schwartz. *Valuing American Options by Simulation: A Simple Least Squares Approach* (2001, *Review of Financial Studies*, Vol. 14, No. 1, pp. 113–147).



---

## Оценка портфеля

Цена — это то, что ты платишь. Ценность — то, что ты получаешь.

*Уоррен Баффетт*

На данный момент у вас должно сложиться достаточно четкое представление о принципах построения и возможностях аналитического пакета DX. Целиком положившись на метод Монте-Карло, мы добились почти полной модульности.

### *Дисконтирование*

Все необходимые операции риск-нейтрального дисконтирования реализуются экземпляром класса `dx.constant_short_rate`.

### *Релевантные данные*

Все исходные данные и параметры хранятся в (нескольких) экземплярах класса `dx.market_environment`.

### *Объекты моделирования*

Соответствующие факторы риска (базовые активы) моделируются экземплярами одного из трех классов:

- `dx.geometric_brownian_motion`;
- `dx.jump_diffusion`;
- `dx.square_root_diffusion`.

### *Объекты оценки*

Оцениваемые опционы моделируются экземплярами одного из двух классов:

- `dx.valuation_mcs_european`;
- `dx.valuation_mcs_american`.

Не хватает только одного: оценки потенциально сложных *портфелей* опционов. Для этого должны быть соблюдены следующие требования.



### *Отсутствие избыточности*

Каждый фактор риска (базовый актив) моделируется только один раз и потенциально может использоваться многими объектами оценки.

### *Корреляция*

Необходимо учитывать корреляцию между факторами риска.

### *Опционные позиции*

Каждая опционная позиция может состоять из определенного количества опционных контрактов.

Несмотря на принципиальную возможность задавать произвольную валюту для объектов моделирования и оценки, в дальнейшем предполагается, что портфели деноминируются в *единой валюте*. Это существенно упрощает агрегацию стоимостей в пределах портфеля, позволяя абстрагироваться от валютных курсов и связанных с ними рисков.

В этой главе мы рассмотрим два новых класса: один — более простой, для моделирования *деривативной позиции*, а другой — более сложный, для моделирования и оценки *портфеля деривативов*. В главе рассматриваются следующие темы.

### *Деривативные позиции*

В этом разделе описан класс моделирования единственной деривативной позиции.

### *Портфели деривативов*

В этом разделе описан базовый класс оценки портфеля, потенциально содержащего множество деривативных позиций.

## **Деривативные позиции**

По сути, *деривативная позиция* — это просто комбинация объекта оценки и параметра, задающего количество моделируемых инструментов.

### **Класс деривативной позиции**

Ниже приведен код класса, моделирующего деривативную позицию. Это фактически контейнер данных и объектов, который снабжен методом `get_info()`, выводящим информацию о содержимом контейнера.

```

#
# Пакет DX
#
# Портфель -- класс деривативной позиции
#
# derivatives_position.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#

class derivatives_position(object):
    ''' Класс моделирования деривативной позиции

    Атрибуты
    =====
    name: строка
        Имя объекта
    quantity: float
        Количество активов/деривативов, образующих позицию
    underlying: строка
        Название базового актива или фактора риска для дериватива
    mar_env: экземпляр класса market_environment
        Константы, списки и кривые, релевантные
        для класса valuation_class
    otype: строка
        Используемый класс оценки
    payoff_func: строка
        Уравнение выплат по деривативу в синтаксисе Python

    Методы
    =====
    get_info:
        Выводит информацию о деривативной позиции
    ...

    def __init__(self, name, quantity, underlying, mar_env,
                  otype, payoff_func):
        self.name = name
        self.quantity = quantity
        self.underlying = underlying
        self.mar_env = mar_env
        self.otype = otype
        self.payoff_func = payoff_func

```

```

def get_info(self):
    print('NAME')
    print(self.name, '\n')
    print('QUANTITY')
    print(self.quantity, '\n')
    print('UNDERLYING')
    print(self.underlying, '\n')
    print('MARKET ENVIRONMENT')
    print('\n**Constants**')
    for key, value in self.mar_env.constants.items():
        print(key, value)
    print('\n**Lists**')
    for key, value in self.mar_env.lists.items():
        print(key, value)
    print('\n**Curves**')
    for key in self.mar_env.curves.items():
        print(key, value)
    print('\nOPTION TYPE')
    print(self.otype, '\n')
    print('PAYOFF FUNCTION')
    print(self.payoff_func)

```

Для создания деривативной позиции понадобится следующая информация.

**name**

Название позиции.

**quantity**

Количество опционов/деривативов.

**underlying**

Экземпляр класса моделирования в качестве фактора риска.

**mar\_env**

Экземпляр класса `dx.market_environment`.

**otype**

Строка "European" или "American".

**payoff\_func**

Строка, задающая функцию выплат.

## Пример использования

Применение класса показано в следующем интерактивном сеансе. Сначала необходимо определить объект моделирования (потребуется не вся информация, а только самая важная).

```
In [99]: from dx_valuation import *
```

```
In [100]: me_gbm = market_environment('me_gbm',  
                                       t.datetime(2020, 1, 1)) ❶
```

```
In [101]: me_gbm.add_constant('initial_value', 36.) ❶  
          me_gbm.add_constant('volatility', 0.2) ❶  
          me_gbm.add_constant('currency', 'EUR') ❶
```

```
In [102]: me_gbm.add_constant('model', 'gbm') ❷
```

❶ Объект `dx.market_environment` для базового актива.

❷ Необходимо указать тип модели.

Для деривативной позиции тоже не требуется “полный” объект `dx.market_environment`. Вся недостающая информация предоставляется позже (на этапе оценки портфеля).

```
In [103]: from derivatives_position import derivatives_position
```

```
In [104]: me_am_put = market_environment('me_am_put',  
                                          dt.datetime(2020, 1, 1)) ❶
```

```
In [105]: me_am_put.add_constant('maturity',  
                                  dt.datetime(2020, 12, 31)) ❶  
          me_am_put.add_constant('strike', 40.) ❶  
          me_am_put.add_constant('currency', 'EUR') ❶
```

```
In [106]: payoff_func = 'np.maximum(strike - instrument_values, 0)' ❷
```

```
In [107]: am_put_pos = derivatives_position(name='am_put_pos',  
                                           quantity=3,  
                                           underlying='gbm',  
                                           mar_env=me_am_put,  
                                           otype='American',  
                                           payoff_func=payoff_func) ❸
```

```

In [108]: am_put_pos.get_info()
NAME
am_put_pos

QUANTITY
3

UNDERLYING
gbm

MARKET ENVIRONMENT

**Constants**
maturity 2020-12-31 00:00:00
strike 40.0
currency EUR

**Lists**

**Curves**

OPTION TYPE
American

PAYOFF FUNCTION
np.maximum(strike - instrument_values, 0)

```

- ❶ Объект `dx.market_environment` для дериватива.
- ❷ Функция выплат по деривативу.
- ❸ Инициализация объекта `derivatives_position`.

## Портфели деривативов

Применительно к портфелю *релевантный рынок* состоит из релевантных факторов риска (базовых активов), их корреляций, а также оцениваемых деривативов и деривативных позиций. С теоретической точки зрения последующий анализ проводится для общей модели финансового рынка  $\mathcal{M}$ , которую мы определили в главе 17 и к которой применяется фундаментальная теорема ценообразования финансовых активов<sup>1</sup>.

---

<sup>1</sup> На практике выбранный здесь подход иногда называют *глобальной оценкой* (см. [3]).

## Класс портфеля

Ниже приведен код достаточно сложного класса, реализующего оценку портфеля на основе фундаментальной теоремы ценообразования финансовых активов с учетом множества релевантных факторов риска и множества деривативных позиций. Все необходимые пояснения даются в виде комментариев в коде.

```
#
# Пакет DX
#
# Портфель -- класс портфеля деривативов
#
# derivatives_portfolio.py
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd

from dx_valuation import *

# Типы моделей, применяемых для учета факторов риска
models = {'gbm': geometric_brownian_motion,
          'jd': jump_diffusion,
          'srd': square_root_diffusion}

# Допустимые типы опционов
otypes = {'European': valuation_mcs_european,
          'American': valuation_mcs_american}

class derivatives_portfolio(object):
    ''' Класс моделирования и оценки портфелей деривативных позиций

    Атрибуты
    =====
    name: строка
           Имя объекта
    positions: словарь
           Словарь позиций (экземпляров класса derivatives_position)
    val_env: экземпляр класса market_environment
           Рыночная среда для оценки портфеля
```

**assets:** словарь

*Словарь рыночных сред для базовых активов*

**correlations:** список

*Корреляции активов*

**fixed\_seed:** булево значение

*Флаг фиксирования затравочного значения  
в генераторе случайных чисел*

**Методы**

=====

**get\_positions:**

*Выводит информацию о позициях портфеля*

**get\_statistics:**

*Возвращает объект DataFrame со статистическими  
показателями портфеля*

'''

```
def __init__(self, name, positions, val_env, assets,
              correlations=None, fixed_seed=False):
    self.name = name
    self.positions = positions
    self.val_env = val_env
    self.assets = assets
    self.underlyings = set()
    self.correlations = correlations
    self.time_grid = None
    self.underlying_objects = {}
    self.valuation_objects = {}
    self.fixed_seed = fixed_seed
    self.special_dates = []
    for pos in self.positions:
        # Определение самой ранней начальной даты
        self.val_env.constants['starting_date'] = \
            min(self.val_env.constants['starting_date'],
                positions[pos].mar_env.pricing_date)
        # Определение самой поздней релевантной даты
        self.val_env.constants['final_date'] = \
            max(self.val_env.constants['final_date'],
                positions[pos].mar_env.constants['maturity'])
        # Сбор всех базовых активов и добавление их
        # в множество (позволяет избежать дублирования)
        self.underlyings.add(positions[pos].underlying)
```

```

# Создание общей сетки дат
start = self.val_env.constants['starting_date']
end = self.val_env.constants['final_date']
time_grid = pd.date_range(start=start, end=end,
    freq=self.val_env.constants['frequency']).to_pydatetime()
time_grid = list(time_grid)
for pos in self.positions:
    maturity_date = \
        positions[pos].mar_env.constants['maturity']
    if maturity_date not in time_grid:
        time_grid.insert(0, maturity_date)
        self.special_dates.append(maturity_date)
if start not in time_grid:
    time_grid.insert(0, start)
if end not in time_grid:
    time_grid.append(end)
# Удаление повторяющихся записей
time_grid = list(set(time_grid))
# Сортировка дат в объекте time_grid
time_grid.sort()
self.time_grid = np.array(time_grid)
self.val_env.add_list('time_grid', self.time_grid)

if correlations is not None:
    # Обработка корреляций
    ul_list = sorted(self.underlyings)
    correlation_matrix = np.zeros((len(ul_list),
        len(ul_list)))
    np.fill_diagonal(correlation_matrix, 1.0)
    correlation_matrix = pd.DataFrame(correlation_matrix,
        index=ul_list,
        columns=ul_list)

    for i, j, corr in correlations:
        corr = min(corr, 0.9999999999999999)
        # Заполнение корреляционной матрицы
        correlation_matrix.loc[i, j] = corr
        correlation_matrix.loc[j, i] = corr
    # Определение матрицы Холецкого
    cholesky_matrix = np.linalg.cholesky(np.array(
        correlation_matrix))
    # Словарь с индексными позициями среза в массиве
    # случайных чисел, который будет использован
    # для соответствующего базового актива

```



```

rn_set = {asset: ul_list.index(asset)
          for asset in self.underlyings}

# Массив случайных чисел, используемый
# для всех базовых активов (в случае корреляции)
random_numbers = sn_random_numbers((len(rn_set),
                                     len(self.time_grid),
                                     self.val_env.constants['paths']),
                                     fixed_seed=self.fixed_seed)

# Добавление созданных массивов в среду оценки,
# которая будет совместно использоваться всеми
# базовыми активами
self.val_env.add_list('cholesky_matrix', cholesky_matrix)
self.val_env.add_list('random_numbers', random_numbers)
self.val_env.add_list('rn_set', rn_set)

for asset in self.underlyings:
    # Выбор рыночной среды для актива
    mar_env = self.assets[asset]
    # Добавление среды оценки в рыночную среду
    mar_env.add_environment(val_env)
    # Выбор правильного класса моделирования
    model = models[mar_env.constants['model']]
    # Создание объекта моделирования
    if correlations is not None:
        self.underlying_objects[asset] = model(asset,
                                                  mar_env, corr=True)
    else:
        self.underlying_objects[asset] = model(asset,
                                                  mar_env, corr=False)

for pos in positions:
    # Выбор правильного класса оценки
    # ('European' или 'American')
    val_class = otypes[positions[pos].otype]
    # Выбор рыночной среды и добавление среды оценки
    mar_env = positions[pos].mar_env
    mar_env.add_environment(self.val_env)
    # Создание объекта оценки
    self.valuation_objects[pos] = \
        val_class(name=positions[pos].name,
                  mar_env=mar_env,
                  underlying=self.underlying_objects[

```

```

        positions[pos].underlying],
        payoff_func=positions[pos].payoff_func)

def get_positions(self):
    ''' Вспомогательный метод для получения сведений
    обо всех деривативных позициях в портфеле. '''
    for pos in self.positions:
        bar = '\n' + 50 * '-'
        print(bar)
        self.positions[pos].get_info()
        print(bar)

def get_statistics(self, fixed_seed=False):
    ''' Возвращает статистическую информацию о портфеле. '''
    res_list = []
    # Проход по всем позициям портфеля
    for pos, value in self.valuation_objects.items():
        p = self.positions[pos]
        pv = value.present_value(fixed_seed=fixed_seed)
        res_list.append([
            p.name,
            p.quantity,
            # Вычисление всех дисконтированных
            # стоимостей отдельного инструмента
            pv,
            value.currency,
            # Стоимость отдельного инструмента,
            # умноженная на количество
            pv * p.quantity,
            # Вычисление дельты позиции
            value.delta() * p.quantity,
            # Вычисление вего позиции
            value.vega() * p.quantity,
        ])
    # Генерирование объекта DataFrame, содержащего все результаты
    res_df = pd.DataFrame(res_list,
                           columns=['name', 'quant.', 'value',
                                    'curr.', 'pos_value', 'pos_delta',
                                    'pos_vega'])

    return res_df

```



## Преимущества ООП

Класс `dx.derivatives_portfolio` иллюстрирует преимущества объектно-ориентированного подхода, рассмотренного в главе 6. С одной стороны, он содержит достаточно много кода, но, с другой стороны, он решает достаточно сложную задачу и обеспечивает необходимую гибкость, поддерживая самые разные сценарии использования. Сложно представить, как можно было бы сделать то же самое без ООП и классов Python.

## Пример использования

В контексте пакета DX возможности моделирования ограничены комбинацией классов моделирования и оценки. Всего есть шесть возможных вариантов.

```
models = {'gbm' : geometric_brownian_motion,
          'jd' : jump_diffusion
          'srd': square_root_diffusion}
```

```
otypes = {'European' : valuation_mcs_european,
          'American' : valuation_mcs_american}
```

В следующем интерактивном примере определяются две разные деривативные позиции, которые затем объединяются в портфель.

В предыдущем разделе мы разработали класс `derivatives_position`, для которого были созданы объекты `am_put_pos` и `me_gbm`. Чтобы продемонстрировать возможности класса `derivatives_portfolio`, мы определим дополнительный базовый актив и дополнительную опционную позицию. Сначала создадим объект `dx.jump_diffusion`.

```
In [109]: me_jd = market_environment('me_jd', me_gbm.pricing_date)
```

```
In [110]: me_jd.add_constant('lambda', 0.3) ❶
          me_jd.add_constant('mu', -0.75)
          me_jd.add_constant('delta', 0.1)
          me_jd.add_environment(me_gbm) ❷
```

```
In [111]: me_jd.add_constant('model', 'jd') ❸
```

- ❶ Добавление параметров прыжковой диффузии.
- ❷ Добавление других параметров объекта `me_gbm`.
- ❸ Это необходимо для оценки портфеля.

Далее на основе нового объекта моделирования создадим объект европейского опциона.

```
In [112]: me_eur_call = market_environment('me_eur_call',
                                             me_jd.pricing_date)

In [113]: me_eur_call.add_constant('maturity',
                                     dt.datetime(2020, 6, 30))
me_eur_call.add_constant('strike', 38.)
me_eur_call.add_constant('currency', 'EUR')

In [114]: payoff_func = 'np.maximum(maturity_value - strike, 0)'

In [115]: eur_call_pos = derivatives_position(
    name='eur_call_pos',
    quantity=5,
    underlying='jd',
    mar_env=me_eur_call,
    otype='European',
    payoff_func=payoff_func)
```

С точки зрения портфеля релевантный рынок описывается объектами `underlyings` и `positions`. На данный момент корреляция между базовыми активами не учитывается. Последний шаг, который необходимо выполнить перед созданием объекта `derivatives_portfolio`, — формирование рыночной среды (объект `dx.market_environment`) для оценки портфеля.

```
In [116]: underlyings = {'gbm': me_gbm, 'jd' : me_jd} ❶
positions = {'am_put_pos' : am_put_pos,
             'eur_call_pos' : eur_call_pos} ❷

In [117]: csr = constant_short_rate('csr', 0.06) ❸

In [118]: val_env = market_environment('general',
                                       me_gbm.pricing_date)
val_env.add_constant('frequency', 'W')
val_env.add_constant('paths', 25000)
val_env.add_constant('starting_date', val_env.pricing_date)
val_env.add_constant('final_date', val_env.pricing_date) ❹
val_env.add_curve('discount_curve', csr) ❺

In [119]: from derivatives_portfolio import derivatives_portfolio

In [120]: portfolio = derivatives_portfolio()
```

```
name='portfolio',
positions=positions,
val_env=val_env,
assets=underlyings,
fixed_seed=False) ⑤
```

- ① Учитываемые факторы риска.
- ② Позиции, включаемые в портфель.
- ③ Уникальный объект дисконтирования для оценки портфеля.
- ④ Параметр `final_date` еще не известен, поэтому предварительно используется значение `pricing_date`.
- ⑤ Создание объекта `derivatives_portfolio`.

Теперь можно легко получить статистическую сводку по только что созданному объекту `derivatives_portfolio`. Также можно вычислить *суммарные* значения стоимости позиций, дельты и веги. В данном портфеле дельта относительно маленькая (почти нейтральная), а вега достаточно большая.

```
In [121]: %time portfolio.get_statistics(fixed_seed=False)
CPU times: user 4.68 s, sys: 409 ms, total: 5.09 s
Wall time: 14.5 s
```

Out[121]:

	name	quant.	value	curr.	pos_value \
0	am_put_pos	3	4.458891	EUR	13.376673
1	eur_call_pos	5	2.828634	EUR	14.143170

	pos_delta	pos_vega
0	-2.0430	31.7850
1	3.2525	42.2655

```
In [122]: portfolio.get_statistics(fixed_seed=False)[
           ['pos_value', 'pos_delta', 'pos_vega']].sum() ①
```

```
Out[122]: pos_value 27.502731
           pos_delta 1.233500
           pos_vega 74.050500
           dtype: float64
```

```
In [123]: portfolio.get_positions() ②
```

```
In [124]: portfolio.valuation_objects['am_put_pos'].present_value() ③
```

```
Out[124]: 4.453187
```

```
In [125]: portfolio.valuation_objects['eur_call_pos'].delta() ④  
Out[125]: 0.6514
```

- ① Агрегация значений отдельных позиций.
- ② Этот метод возвращает достаточно длинную сводку обо всех позициях портфеля.
- ③ Оценка дисконтированной стоимости отдельной позиции.
- ④ Оценка дельты отдельной позиции.

Оценка портфеля деривативов строится на основе предположения об отсутствии корреляции между учитываемыми факторами риска. В этом легко убедиться, проанализировав две смоделированные траектории (рис. 20.1), по одной для каждого объекта.

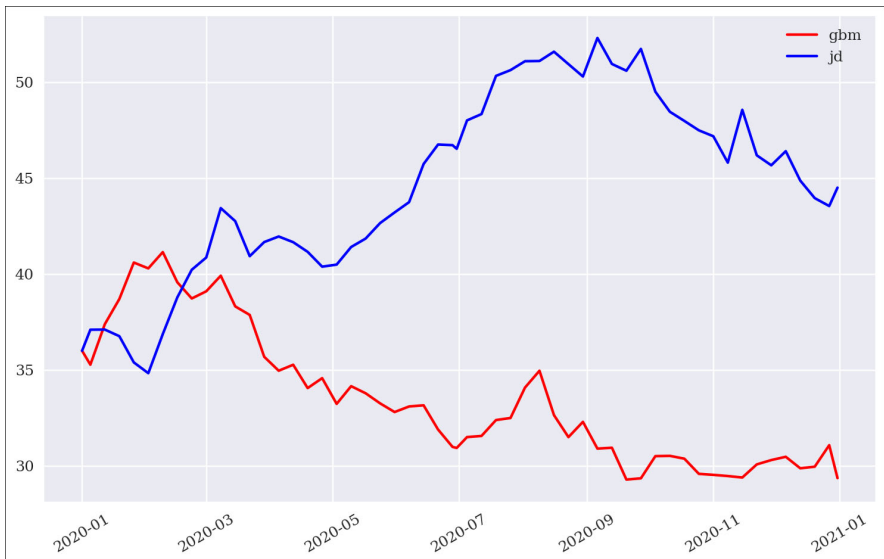


Рис. 20.1. Некоррелированные факторы риска

```
In [126]: path_no = 888  
path_gbm = portfolio.underlying_objects[  
    'gbm'].get_instrument_values(:, path_no)  
path_jd = portfolio.underlying_objects[  
    'jd'].get_instrument_values(:, path_no)
```

```
In [127]: plt.figure(figsize=(10,6))
          plt.plot(portfolio.time_grid, path_gbm, 'r', label='gbm')
          plt.plot(portfolio.time_grid, path_jd, 'b', label='jd')
          plt.xticks(rotation=30)
          plt.legend(loc=0)
```

Теперь рассмотрим ситуацию, когда между двумя факторами риска наблюдается явная корреляция. В данном случае это не оказывает прямого влияния на стоимости отдельных позиций портфеля.

```
In [128]: correlations = [['gbm', 'jd', 0.9]]
```

```
In [129]: port_corr = derivatives_portfolio(
          name='portfolio',
          positions=positions,
          val_env=val_env,
          assets=underlyings,
          correlations=correlations,
          fixed_seed=True)
```

```
In [130]: port_corr.get_statistics()
```

```
Out[130]:
```

	name	quant.	value	curr.	pos_value \
0	am_put_pos	3	4.458556	EUR	13.375668
1	eur_call_pos	5	2.817813	EUR	14.089065

	pos_delta	pos_vega
0	-2.0376	30.8676
1	3.3375	42.2340

Тем не менее в существовании корреляции можно убедиться, визуализировав ту же самую комбинацию траекторий. Оба графика демонстрируют почти параллельное движение (рис. 20.2).

```
In [131]: path_gbm = port_corr.underlying_objects['gbm']. \
          get_instrument_values()[ :, path_no]
          path_jd = port_corr.underlying_objects['jd']. \
          get_instrument_values()[ :, path_no]
```

```
In [132]: plt.figure(figsize=(10, 6))
          plt.plot(portfolio.time_grid, path_gbm, 'r', label='gbm')
          plt.plot(portfolio.time_grid, path_jd, 'b', label='jd')
          plt.xticks(rotation=30)
          plt.legend(loc=0);
```

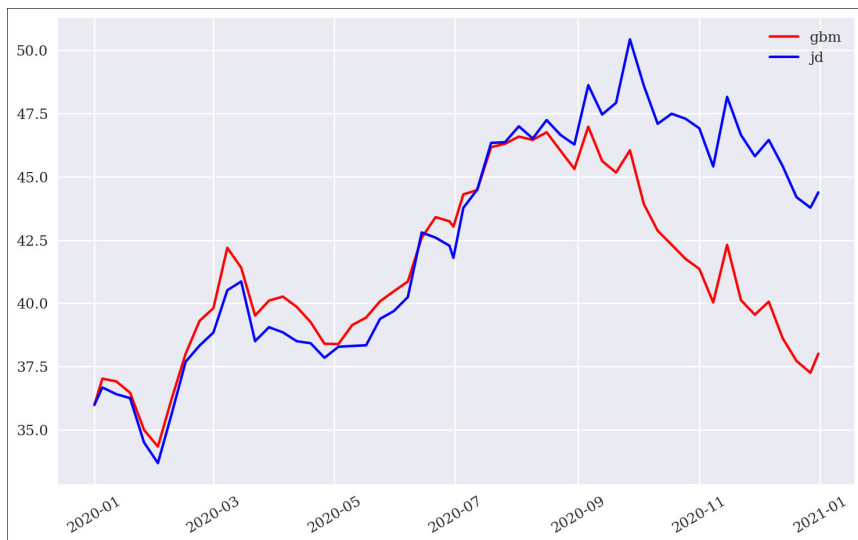


Рис. 20.2. Коррелированные факторы риска

В качестве последнего примера рассмотрим *частотное распределение дисконтированной стоимости портфеля*. Его, как правило, невозможно получить другими способами, например с помощью аналитических формул или биномиальной модели оценки опционов. Если задать параметр `full` метода `present_value()` равным `True`, то будет получен полный набор дисконтированных стоимостей по опционной позиции.

```
In [133]: pv1 = 5 * port_corr.valuation_objects['eur_call_pos'].\
           present_value(full=True)[1]
```

pv1

```
Out[133]: array([ 0.          , 39.71423714, 24.90720272, ...,
                  0.          ,  6.42619093,  8.15838265])
```

```
In [134]: pv2 = 3 * port_corr.valuation_objects['am_put_pos'].\
           present_value(full=True)[1]
```

pv2

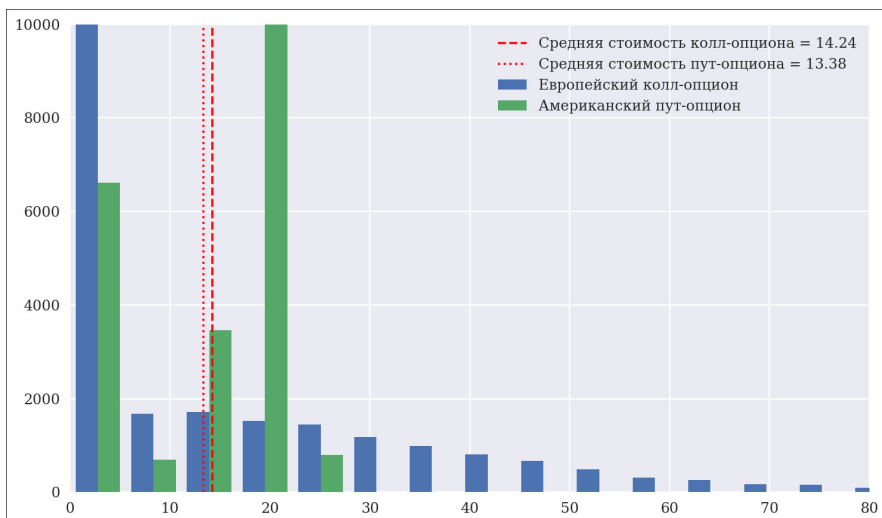
```
Out[134]: array([21.31806027, 10.71952869, 19.89804376, ...,
                  21.39292703, 17.59920608,  0.          ])
```

Для начала сравним частотные распределения двух позиций. Как можно увидеть на рис. 20.3, их профили выплат заметно отличаются. Учтите, что для наглядности на осях X и Y отображаются неполные наборы значений.

```
In [135]: plt.figure(figsize=(10, 6))
```



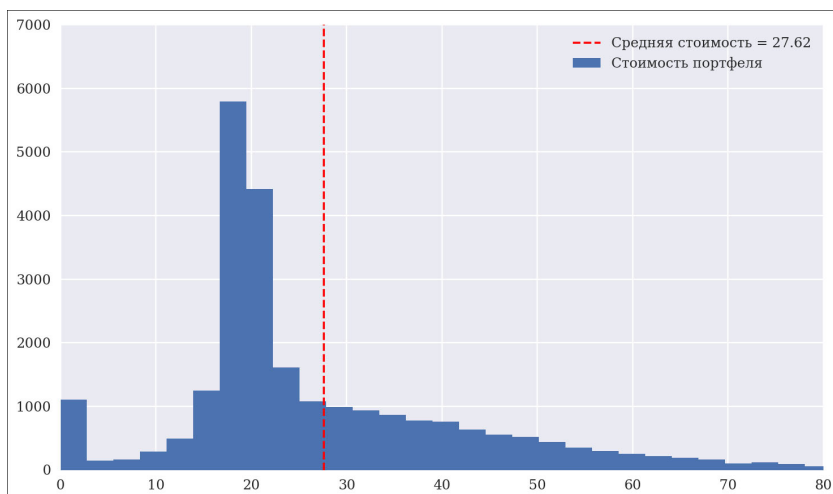
```
plt.hist([pv1, pv2], bins=25,
        label=['Европейский колл-опцион',
               'Американский пут-опцион']);
plt.axvline(pv1.mean(), color='r', ls='dashed',
            lw=1.5, label='Средняя стоимость
колл-опциона = %4.2f' % pv1.mean())
plt.axvline(pv2.mean(), color='r', ls='dotted',
            lw=1.5, label='Средняя стоимость
пут-опциона = %4.2f' % pv2.mean())
plt.xlim(0, 80); plt.ylim(0, 10000)
plt.legend();
```



*Рис. 20.3. Частотное распределение дисконтированных стоимостей двух позиций*

Наконец, на рис. 20.4 показано частотное распределение дисконтированной стоимости портфеля опционов. На диаграмме четко проявляется эффект смещения при комбинировании опционов “колл” и “пут”.

```
In [136]: pvs = pv1 + pv2
plt.figure(figsize=(10, 6))
plt.hist(pvs, bins=50, label='Стоимость портфеля');
plt.axvline(pvs.mean(), color='r', ls='dashed',
            lw=1.5, label='Средняя стоимость = %4.2f' %
pvs.mean())
plt.xlim(0, 80); plt.ylim(0, 7000)
plt.legend();
```



*Рис. 20.4. Частотное распределение дисконтированной стоимости портфеля*

Как корреляция между двумя факторами риска влияет на риск всего портфеля, если рассматривать стандартное отклонение дисконтированных стоимостей? Ответ можно получить, выполнив следующие две оценки.

In [137]: `pvs.std()` ❶

Out[137]: 16.723724772741118

```
In [138]: pv1 = (5 * portfolio.valuation_objects['eur_call_pos'].
            present_value(full=True)[1])
            pv2 = (3 * portfolio.valuation_objects['am_put_pos'].
            present_value(full=True)[1])
            (pv1 + pv2).std() ❷
```

Out[138]: 21.80498672323975

- ❶ Стандартное отклонение стоимости портфеля с учетом корреляции.
- ❷ Стандартное отклонение стоимости портфеля без учета корреляции.

Несмотря на то что среднее значение остается постоянным (исключая числовые изменения), вполне очевидно, что корреляция значительно снижает риск портфеля, по крайней мере, при таком способе анализа. Опять-таки, подобный вывод невозможно сделать, применяя другие способы оценки портфеля.

## Резюме

В этой главе рассматривались принципы оценки портфеля деривативных позиций с учетом множества (потенциально коррелированных) факторов риска. Для этого мы разработали класс `derivatives_position`, моделирующий деривативную позицию. Но основной нашей целью был класс `derivatives_portfolio`, который, в частности, позволяет решать следующие задачи:

- *учет корреляции* между факторами риска (класс генерирует согласованный набор случайных чисел для моделирования всех факторов риска);
- *создание объектов моделирования* на основе отдельных рыночных сред и общей среды оценки, а также заданных деривативных позиций;
- *генерирование статистической сводки портфеля* на основе всех сделанных предположений, учитываемых факторов риска и параметров деривативных позиций.

Представленные в этой главе примеры иллюстрируют моделирование только простых портфелей деривативов, которыми можно управлять с помощью разработанного нами пакета `DX` и класса `derivatives_portfolio`. Логично было бы расширить пакет `DX`, включив в него более сложные финансовые модели, например модель стохастической волатильности, и классы мультирисковой оценки, позволяющие моделировать деривативы, которые зависят от нескольких факторов риска (в частности, европейский корзинный опцион или американский максимальный колл-опцион). На данном этапе модульный подход к моделированию на основе ООП и фундаментальной теоремы ценообразования финансовых активов (схема “глобальной оценки”) дает свои преимущества: безызыточное моделирование факторов риска и учет корреляции между ними окажут прямое влияние на стоимости мультирисковых деривативов и значения их греческих коэффициентов.

Ниже приведена финальная версия оболочечного модуля, объединяющего все применяемые в пакете `DX` команды импорта.

```
#  
# Пакет DX  
#  
# Все компоненты  
#  
# dx_package.py  
#  
# Python for Finance, 2nd ed.  
# (c) Dr. Yves J. Hilpisch
```

```
#
from dx_valuation import *
from derivatives_position import derivatives_position
from derivatives_portfolio import derivatives_portfolio
```

А вот полный вариант файла `__init__.py`, хранящегося в папке `dx`.

```
#
# Пакет DX
#
# Пакетный файл
#
# __init__.py
#
import numpy as np
import pandas as pd
import datetime as dt

# Базовые инструменты
from get_year_deltas import get_year_deltas
from constant_short_rate import constant_short_rate
from market_environment import market_environment
from plot_option_stats import plot_option_stats

# Моделирование
from sn_random_numbers import sn_random_numbers
from simulation_class import simulation_class
from geometric_brownian_motion import geometric_brownian_motion
from jump_diffusion import jump_diffusion
from square_root_diffusion import square_root_diffusion

# Оценка деривативов
from valuation_class import valuation_class
from valuation_mcs_european import valuation_mcs_european
from valuation_mcs_american import valuation_mcs_american

# Оценка портфеля
from derivatives_position import derivatives_position
from derivatives_portfolio import derivatives_portfolio
```

## Дополнительные ресурсы

Как и в предыдущих главах, посвященных разработке пакета DX, исчерпывающим руководством по применению метода Монте-Карло в финансовом моделировании послужит книга Глассермана [1]. У Хилпиша [2] рассматривается реализация наиболее важных алгоритмов Монте-Карло на Python.

Впрочем, когда дело касается согласованной оценки (сложных) портфелей деривативов на основе метода Монте-Карло, серьезных исследовательских работ практически нет. На концептуальном уровне данная тема рассмотрена разве что в короткой статье Альбанезе, Жимоне и Уайта [3]. У этих же авторов есть также более детальная статья [4].

1. Glasserman, Paul. *Monte Carlo Methods in Financial Engineering* (2004, Springer).
2. Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).
3. Albanese, Claudio, Guillaume Gimonet and Steve White. *Towards a Global Valuation Model* (2010, *Risk Magazine*, Vol. 23, No. 5, pp. 68–71; [http://bit.ly/risk\\_may\\_2010](http://bit.ly/risk_may_2010)).
4. Albanese, Claudio, Guillaume Gimonet and Steve White. *Global Valuation and Dynamic Risk Management* (2010; [http://bit.ly/global\\_valuation](http://bit.ly/global_valuation)).

---

## Оценка на основе рыночных данных

Мы сталкиваемся с крайней волатильностью.

*Карлос Гон*

Основной задачей анализа деривативов является *рыночная оценка опционов*, по которым нет ликвидных торгов. Для этого модель оценки сначала калибруется по рыночным котировкам ликвидно торгуемых опционов, а уже затем откалиброванная модель применяется для оценки неторгуемых опционов.

В этой главе мы рассмотрим пример на основе пакета DX и покажем, что данный пакет, который мы пошагово разработали в предыдущих четырех главах, подходит для реализации рыночной оценки. В примере будет использоваться фондовый индекс DAX, включающий акции 30 крупнейших компаний Германии. По этому индексу проводятся рыночные торги европейскими опционами “пут” и “колл”.

В главе рассматриваются следующие темы.

### *Данные опционов*

Для выполнения расчетов нам понадобятся два типа данных: собственно фондовый индекс DAX и стоимости связанных с ним ликвидно торгуемых европейских опционов.

### *Калибровка модели*

Для рыночной оценки неторгуемых опционов сначала необходимо откалибровать выбранную модель по котировкам опционов таким образом, чтобы модель с оптимальными параметрами как можно точнее соответствовала существующим рыночным ценам.

### *Оценка портфеля*

Создав откалиброванную модель рынка и получив биржевые данные по индексу DAX, можно приступить к решению основной задачи — моделированию и оценке неторгуемых опционов. На этом этапе также оцениваются все основные факторы риска на уровне как отдельных позиций, так и всего портфеля.

Данные опционов и фондового индекса DAX, используемые в примерах этой главы, получены с помощью программного интерфейса Thomson Reuters Eikon.

## Данные опционов

Для начала импортируем необходимые библиотеки и модули и зададим базовые параметры.

```
In [1]: import numpy as np
import pandas as pd
import datetime as dt
```

```
In [2]: from pylab import mpl, plt
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

```
In [3]: import sys
sys.path.append('../dx')
```

Данные будут считываться из файла *tr\_eikon\_option\_data.csv*, который создается в разделе “Код Python”. С помощью библиотеки *pandas* данные обрабатываются таким образом, что даты представляются объектами *pd.Timestamp*.

```
In [4]: dax = pd.read_csv('../source/tr_eikon_option_data.csv',
index_col=0) ❶
```

```
In [5]: for col in ['CF_DATE', 'EXPIR_DATE']:
dax[col] = dax[col].apply(lambda date:
pd.Timestamp(date)) ❷
```

```
In [6]: dax.info() ❸
<class 'pandas.core.frame.DataFrame'>
Int64Index: 115 entries, 0 to 114
Data columns (total 7 columns):
Instrument    115 non-null object
CF_DATE       115 non-null datetime64[ns]
EXPIR_DATE    114 non-null datetime64[ns]
PUTCALLIND    114 non-null object
STRIKE_PRC    114 non-null float64
CF_CLOSE      115 non-null float64
IMP_VOLT      114 non-null float64
```

```
dtypes: datetime64[ns](2), float64(3), object(2)
memory usage: 7.2+ KB
```

```
In [7]: dax.set_index('Instrument').head(7) ❸
```

```
Out[7]:
```

	CF_DATE	EXPIR_DATE	PUTCALLIND \
<b>Instrument</b>			
.GDAXI	2018-04-27	NaN	NaN
GDAX105000G8.EX	2018-04-27	2018-07-20	CALL
GDAX105000S8.EX	2018-04-27	2018-07-20	PUT
GDAX108000G8.EX	2018-04-27	2018-07-20	CALL
GDAX108000S8.EX	2018-04-26	2018-07-20	PUT
GDAX110000G8.EX	2018-04-27	2018-07-20	CALL
GDAX110000S8.EX	2018-04-27	2018-07-20	PUT

	STRIKE_PRC	CF_CLOSE	IMP_VOLT
<b>Instrument</b>			
.GDAXI	NaN	12500.47	NaN
GDAX105000G8.EX	10500.0	2040.80	23.59
GDAX105000S8.EX	10500.0	32.00	23.59
GDAX108000G8.EX	10800.0	1752.40	22.02
GDAX108000S8.EX	10800.0	43.80	22.02
GDAX110000G8.EX	11000.0	1562.80	21.00
GDAX110000S8.EX	11000.0	54.50	21.00

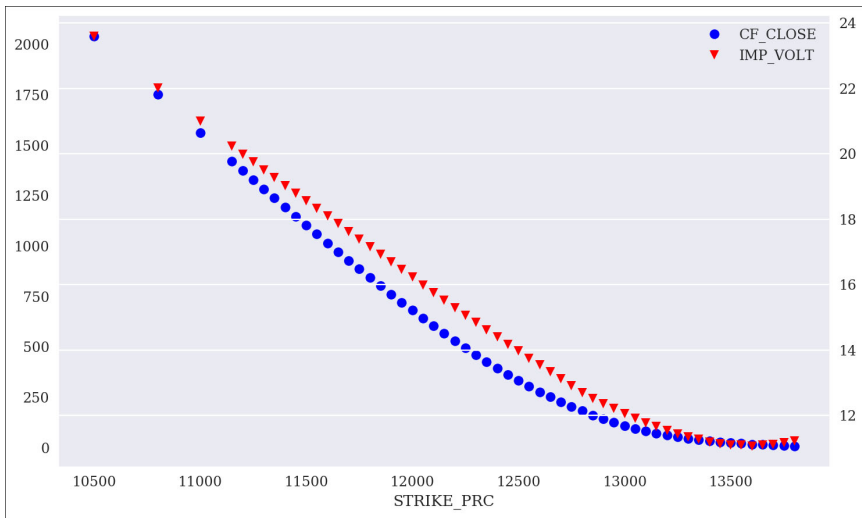
- ❶ Считывание данных с помощью функции `pd.read_csv()`.
- ❷ Обработка двух столбцов с датами.
- ❸ Полученный объект `DataFrame`.

В следующем коде релевантный уровень индекса DAX сохраняется в отдельной переменной, а также создаются два новых объекта `DataFrame`: один — для колл-опционов, другой — для пут-опционов. На рис. 21.1 показаны рыночные котировки колл-опционов и их ожидаемые волатильности<sup>1</sup>.

---

<sup>1</sup> Ожидаемая волатильность опциона — это значение волатильности, подстановка которого в формулу Блэка — Шоулза — Мертона ([1] и [3]) приводит к получению рыночной котировки опциона.





*Рис. 21.1. Рыночные котировки и ожидаемые волатильности европейских колл-опционов индекса DAX*

```
In [8]: initial_value = dax.iloc[0]['CF_CLOSE'] ❶
```

```
In [9]: calls = dax[dax['PUTCALLIND'] == 'CALL'].copy() ❷
```

```
puts = dax[dax['PUTCALLIND'] == 'PUT '].copy() ❷
```

```
In [10]: calls.set_index('STRIKE_PRC')[['CF_CLOSE',
      'IMP_VOLT']].plot(secondary_y='IMP_VOLT',
      style=['bo', 'rv'], figsize=(10, 6), mark_right=False);
```

❶ Запись релевантного уровня индекса в переменную `initial_value`.

❷ Разделение данных на два объекта `DataFrame` для опционов “пут” и “колл”.

Рыночные котировки и ожидаемые волатильности пут-опционов приведены на рис. 21.2.

```
In [11]: ax = puts.set_index('STRIKE_PRC')[['CF_CLOSE',
      'IMP_VOLT']].plot(secondary_y='IMP_VOLT',
      style=['bo', 'rv'], figsize=(10, 6), mark_right=False)
ax.get_legend().set_bbox_to_anchor((0.25, 0.5));
```

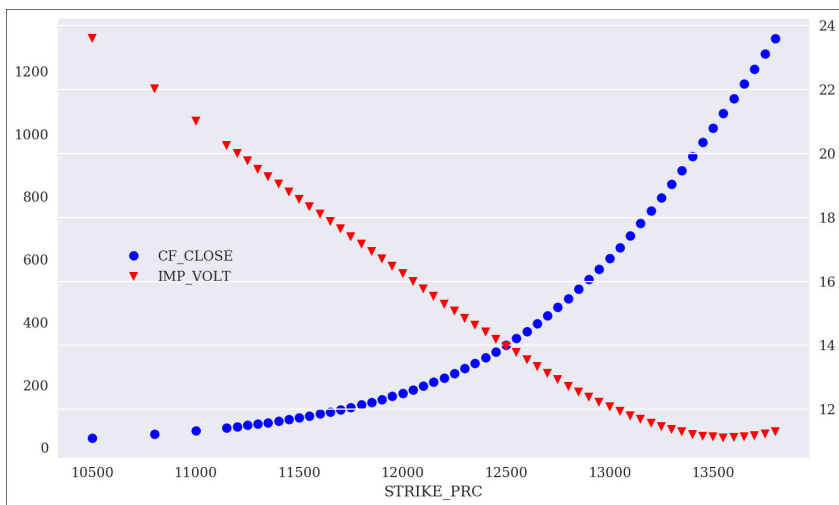


Рис. 21.2. Рыночные котировки и ожидаемые волатильности европейских пут-опционов индекса DAX

## Калибровка модели

В этом разделе мы соберем все необходимые рыночные данные, смоделируем европейские опционы индекса DAX и выполним калибровку модели.

### Релевантные рыночные данные

Калибровка модели обычно выполняется на подмножестве доступных рыночных котировок опционов<sup>2</sup>. Поэтому в следующем коде мы отбираем только те европейские колл-опционы, страйк-цена которых относительно близка к текущему уровню индекса (рис. 21.3). Другими словами, в расчет принимаются только те колл-опционы, которые не показывают чрезмерную прибыльность или убыточность.

```
In [12]: limit = 500 ❶
```

```
In [13]: option_selection = calls[abs(calls['STRIKE_PRC'] -  
                                     initial_value) < limit].copy() ❷
```

```
In [14]: option_selection.info() ❸
```

<sup>2</sup> Подробнее об этом рассказано в главе 11 книги Хилпиша [2].

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 20 entries, 43 to 81
Data columns (total 7 columns):
Instrument      20 non-null object
CF_DATE         20 non-null datetime64[ns]
EXPIR_DATE      20 non-null datetime64[ns]
PUTCALLIND      20 non-null object
STRIKE_PRC      20 non-null float64
CF_CLOSE        20 non-null float64
IMP_VOLT        20 non-null float64
dtypes: datetime64[ns](2), float64(3), object(2)
memory usage: 1.2+ KB
```

In [15]: `option_selection.set_index('Instrument').tail()` ❸  
 Out[15]:

	CF_DATE	EXPIR_DATE	PUTCALLIND \
<b>Instrument</b>			
GDAX128000G8.EX	2018-04-27	2018-07-20	CALL
GDAX128500G8.EX	2018-04-27	2018-07-20	CALL
GDAX129000G8.EX	2018-04-25	2018-07-20	CALL
GDAX129500G8.EX	2018-04-27	2018-07-20	CALL
GDAX130000G8.EX	2018-04-27	2018-07-20	CALL

	STRIKE_PRC	CF_CLOSE	IMP_VOLT
<b>Instrument</b>			
GDAX128000G8.EX	12800.0	182.4	12.70
GDAX128500G8.EX	12850.0	162.0	12.52
GDAX129000G8.EX	12900.0	142.9	12.36
GDAX129500G8.EX	12950.0	125.4	12.21
GDAX130000G8.EX	13000.0	109.4	12.06

In [16]: `option_selection.set_index('STRIKE_PRC')[['CF_CLOSE',  
 'IMP_VOLT']].plot(secondary_y='IMP_VOLT',  
 style=['bo', 'rv'], figsize=(10, 6), mark_right=False);`

- ❶ Задание параметра `limit` для определения страйк-цены по текущему уровню индекса (условие *денежности* опциона).
- ❷ Отбор (на основе параметра `limit`) европейских колл-опционов, используемых для калибровки модели.
- ❸ Полученный объект `DataFrame`, включающий европейские колл-опционы, которые применяются для калибровки модели.

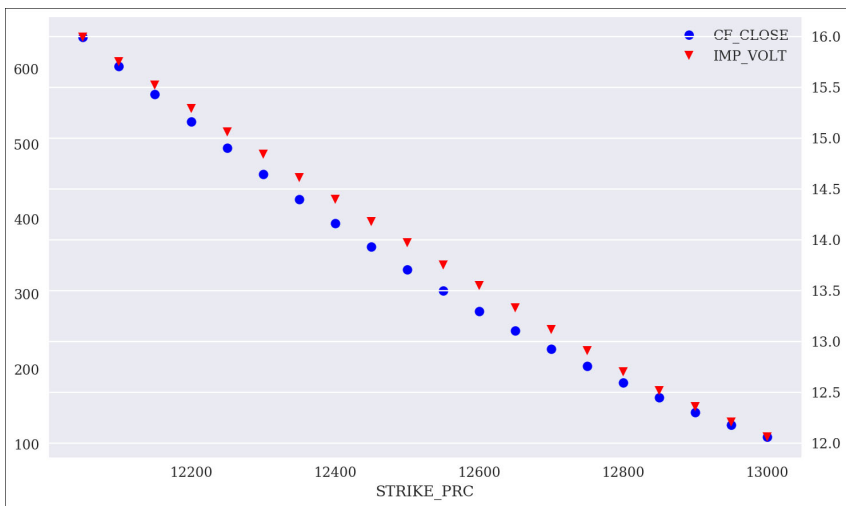


Рис. 21.3. Европейские колл-опционы индекса DAX, применяемые для калибровки модели

## Моделирование опционов

Отобрав релевантные рыночные данные, мы теперь можем применить пакет DX для моделирования европейских колл-опционов. Ниже создается объект `dx.market_environment`, моделирующий рыночную среду индекса DAX.

```
In [17]: from dx_package import *
```

```
In [18]: pricing_date = option_selection['CF_DATE'].max() ❶
```

```
In [19]: me_dax = market_environment('DAX30', pricing_date) ❷
```

```
In [20]: maturity = pd.Timestamp(calls.iloc[0]['EXPIR_DATE']) ❸
```

```
In [21]: me_dax.add_constant('initial_value', initial_value) ❹
me_dax.add_constant('final_date', maturity) ❹
me_dax.add_constant('currency', 'EUR') ❹
```

```
In [22]: me_dax.add_constant('frequency', 'B') ❺
me_dax.add_constant('paths', 10000) ❺
```

```
In [23]: csr = constant_short_rate('csr', 0.01) ❻
me_dax.add_curve('discount_curve', csr) ❻
```

- ❶ Определение начальной даты или даты ценообразования на основе данных опциона.
- ❷ Создание объекта `dx.market_environment`.
- ❸ Определение даты исполнения на основе данных опциона.
- ❹ Добавление базовых параметров модели.
- ❺ Задание параметров моделирования.
- ❻ Добавление объекта `dx.constant_short_rate`.

В следующем коде параметры модели добавляются в класс `dx.jump_diffusion`, и мы получаем соответствующий объект моделирования.

```
In [24]: me_dax.add_constant('volatility', 0.2)
         me_dax.add_constant('lambda', 0.8)
         me_dax.add_constant('mu', -0.2)
         me_dax.add_constant('delta', 0.1)
```

```
In [25]: dax_model = jump_diffusion('dax_model', me_dax)
```

В качестве примера европейского колл-опциона рассмотрим следующий вариант параметризации, в котором страйк-цена задается равной текущему уровню индекса DAX. Это позволяет получить первую оценку стоимости на основе метода Монте-Карло.

```
In [26]: me_dax.add_constant('strike', initial_value) ❶
         me_dax.add_constant('maturity', maturity)
```

```
In [27]: payoff_func = 'np.maximum(maturity_value - strike, 0)' ❷
```

```
In [28]: dax_eur_call = valuation_mcs_european('dax_eur_call',
         dax_model, me_dax, payoff_func) ❸
```

```
In [29]: dax_eur_call.present_value() ❹
```

```
Out[29]: 654.298085
```

- ❶ Параметр `strike` задается равным переменной `initial_value`.
- ❷ Определение функции выплат по европейскому колл-опциону.
- ❸ Создание объекта оценки.
- ❹ Запуск процедуры моделирования и оценки.

Аналогичным образом можно определить объекты оценки для всех европейских колл-опционов из индекса DAX. Единственный изменяемый параметр — это страйк-цена.

```
In [30]: option_models = {} ❶
        for option in option_selection.index:
            strike = option_selection['STRIKE_PRC'].loc[option] ❷
            me_dax.add_constant('strike', strike) ❷
            option_models[strike] = valuation_mcs_european(
                'eur_call_%d' % strike,
                dax_model,
                me_dax,
                payoff_func)
```

❶ Объекты оценки собираются словарь.

❷ Выбор соответствующей страйк-цены и ее переопределение в объекте `dx.market_environment`.

Теперь, когда у нас есть объекты оценки для всех релевантных опционов, напомним функцию `calculate_model_values()`, которая рассчитывает их прогнозируемые стоимости с учетом набора параметров `p0`.

```
In [32]: def calculate_model_values(p0):
        ''' Возвращает стоимости всех релевантных опционов

        Параметры
        =====
        p0: кортеж/список
            Кортеж параметров модели

        Возвращает
        =====
        model_values: словарь
            Словарь моделируемых стоимостей
        '''
        volatility, lamb, mu, delta = p0
        dax_model.update(volatility=volatility, lamb=lamb,
                        mu=mu, delta=delta)
        return {
            strike: model.present_value(fixed_seed=True)
            for strike, model in option_models.items()
        }
```

```
In [33]: calculate_model_values((0.1, 0.1, -0.4, 0.0))
```

```
Out[33]: {12050.0: 611.222524,
          12100.0: 571.83659 ,
          12150.0: 533.595853,
          12200.0: 496.607225,
          12250.0: 460.863233,
          12300.0: 426.543355,
          12350.0: 393.626483,
          12400.0: 362.066869,
          12450.0: 331.877733,
          12500.0: 303.133596,
          12550.0: 275.987049,
          12600.0: 250.504646,
          12650.0: 226.687523,
          12700.0: 204.550609,
          12750.0: 184.020514,
          12800.0: 164.945082,
          12850.0: 147.249829,
          12900.0: 130.831722,
          12950.0: 115.681449,
          13000.0: 101.917351}
```

Функция `calculate_model_values()` применяется на этапе калибровке модели, как будет описано далее.

## Процедура калибровки

В общем случае калибровка модели ценообразования опциона представляет собой задачу выпуклого программирования. Под калибровкой понимают минимизацию некоторой функции ошибок, при этом чаще всего используют *среднеквадратическую ошибку* (Mean-Squared Error — MSE) цен опционов, моделируемых на основе заданных рыночных котировок<sup>3</sup>. Предположим, имеются  $N$  релевантных опционов, для которых известны спрогнозированные стоимости и рыночные котировки. Тогда задачу калибровки модели по рыночным котировкам на основе среднеквадратической ошибки можно описать уравнением 21.1, где  $C_n^*$  и  $C_n^{mod}$  — соответственно рыночная и прогнозируемая цена  $n$ -го опциона, а  $p$  — набор параметров, передаваемых модели в качестве входных данных.

---

<sup>3</sup> Существуют и другие варианты целевых функций для процедуры калибровки; см. главу 11 книги Хилпиша [2].

## Уравнение 21.1. Среднеквадратическая ошибка для калибровки модели

$$\min_p \frac{1}{N} \sum_{n=1}^N \left( C_n^* - C_n^{mod}(p) \right)^2.$$

Данный подход реализован в функции `mean_squared_error()`. Глобальная переменная `i` управляет выводом промежуточных кортежей и рассчитанных значений среднеквадратической ошибки.

In [34]: `i = 0`

```
def mean_squared_error(p0):
    ''' Возвращает среднеквадратическую ошибку с учетом
        параметров модели и рыночных котировок.

        Параметры
        =====
        p0: кортеж/список
            Кортеж параметров модели

        Возвращает
        =====
        MSE: float
            Среднеквадратическая ошибка
    '''
    global i
    model_values = np.array(list(
        calculate_model_values(p0).values())) ❶
    market_values = option_selection['CF_CLOSE'].values ❷
    option_diffs = model_values - market_values ❸
    MSE = np.sum(option_diffs ** 2) / len(option_diffs) ❹
    if i % 75 == 0:
        if i == 0:
            print('%4s %6s %6s %6s %6s --> %6s' %
                  ('i', 'vola', 'lambda', 'mu', 'delta',
                   'MSE'))
            print('%4d %6.3f %6.3f %6.3f %6.3f --> %6.3f' %
                  (i, p0[0], p0[1], p0[2], p0[3], MSE))
        i += 1
    return MSE
```

```
In [35]: mean_squared_error((0.1, 0.1, -0.4, 0.0))❺
        i   vola  lambda      mu   delta -->   MSE
        0   0.100   0.100  -0.400   0.000 --> 728.375
```

Out[35]: 728.3752973715275



- ❶ Получение прогнозируемых стоимостей.
- ❷ Выбор рыночных котировок.
- ❸ Поэлементный расчет разницы между прогнозируемыми стоимостями и рыночными котировками.
- ❹ Вычисление среднеквадратической ошибки.
- ❺ Пример расчетов.

В главе 11 рассматривались функции `spo.brute()` и `spo.fmin()`, которые мы применим для калибровки модели. Сначала выполним глобальную минимизацию на основе диапазонов для четырех параметров модели. Мы получим оптимальную комбинацию параметров, вычисленную путем перебора *методом грубой силы*.

```
In [36]: import scipy.optimize as spo
```

```
In [37]: %%time
i = 0
opt_global = spo.brute(mean_squared_error,
                       ((0.10, 0.201, 0.025),
                        # диапазон волатильности
                        (0.10, 0.80, 0.10),
                        # диапазон интенсивности скачков
                        (-0.40, 0.01, 0.10),
                        # диапазон средней величины скачка
                        (0.00, 0.121, 0.02)),
                       # диапазон вариативности скачков
                       finish=None)
```

i	vola	lambda	mu	delta	--> MSE
0	0.100	0.100	-0.400	0.000	--> 728.375
75	0.100	0.300	-0.400	0.080	--> 5157.513
150	0.100	0.500	-0.300	0.040	--> 12199.386
225	0.100	0.700	-0.200	0.000	--> 6904.932
300	0.125	0.200	-0.200	0.100	--> 855.412
375	0.125	0.400	-0.100	0.060	--> 621.800
450	0.125	0.600	0.000	0.020	--> 544.137
525	0.150	0.100	0.000	0.120	--> 3410.776
600	0.150	0.400	-0.400	0.080	--> 46775.769
675	0.150	0.600	-0.300	0.040	--> 56331.321
750	0.175	0.100	-0.200	0.000	--> 14562.213
825	0.175	0.300	-0.200	0.100	--> 24599.738

```

900 0.175 0.500 -0.100 0.060 --> 19183.167
975 0.175 0.700 0.000 0.020 --> 11871.683
1050 0.200 0.200 0.000 0.120 --> 31736.403
1125 0.200 0.500 -0.400 0.080 --> 130372.718
1200 0.200 0.700 -0.300 0.040 --> 126365.140
CPU times: user 1min 45s, sys: 7.07 s, total: 1min 52s
Wall time: 1min 56s

```

In [38]: `mean_squared_error(opt_global)`

Out[38]: 17.946670038040985

Значения `opt_global` представляют собой лишь промежуточные результаты. Они используются в качестве начальных значений в задаче *локальной минимизации*. А вот значения `opt_local` являются финальными и оптимальными для заданных уровней допуска.

In [39]: `%time`

```

i = 0
opt_local = spo.fmin(mean_squared_error, opt_global,
                    xtol=0.00001, ftol=0.00001,
                    maxiter=200, maxfun=550)

```

```

i   vola  lambda      mu  delta --> MSE
0  0.100  0.200 -0.300  0.000 --> 17.947
75 0.098  0.216 -0.302 -0.001 --> 7.885
150 0.098  0.216 -0.300 -0.001 --> 7.371
Optimization terminated successfully.
    Current function value: 7.371163
    Iterations: 100
    Function evaluations: 188
CPU times: user 15.6 s, sys: 1.03 s, total: 16.6 s
Wall time: 16.7 s

```

In [40]: `i = 0`

```

mean_squared_error(opt_local) ❶
i   vola  lambda      mu  delta --> MSE
0  0.098  0.216 -0.300 -0.001 --> 7.371

```

Out[40]: 7.371162645265256

In [41]: `calculate_model_values(opt_local)` ❷

Out[41]: {12050.0: 647.428189,  
12100.0: 607.402796,  
12150.0: 568.46137,

```

12200.0: 530.703659,
12250.0: 494.093839,
12300.0: 458.718401,
12350.0: 424.650128,
12400.0: 392.023241,
12450.0: 360.728543,
12500.0: 330.727256,
12550.0: 302.117223,
12600.0: 274.98474,
12650.0: 249.501807,
12700.0: 225.678695,
12750.0: 203.490065,
12800.0: 182.947468,
12850.0: 163.907583,
12900.0: 146.259349,
12950.0: 129.909743,
13000.0: 114.852425}

```

- ❶ Среднеквадратическая ошибка при оптимальных значениях параметров.
- ❷ Прогнозируемые стоимости при оптимальных значениях параметров.

Далее мы сравним прогнозируемые стоимости при оптимальных значениях параметров с рыночными котировками. Ошибки оценки вычисляются как абсолютные разницы между спрогнозированными и рыночными значениями, а также как процентные отклонения от рыночных котировок.

```

In [42]: option_selection['MODEL'] = np.array(list(
        calculate_model_values(opt_local).values()))
option_selection['ERRORS_EUR'] = (option_selection[
    'MODEL'] - option_selection['CF_CLOSE'])
option_selection['ERRORS_%'] = (option_selection[
    'ERRORS_EUR'] / option_selection['CF_CLOSE']) * 100

```

```

In [43]: option_selection[['MODEL', 'CF_CLOSE', 'ERRORS_EUR',
    'ERRORS_%']]

```

Out[43]:

	MODEL	CF_CLOSE	ERRORS_EUR	ERRORS_%
43	647.428189	642.6	4.828189	0.751352
45	607.402796	604.4	3.002796	0.496823
47	568.461370	567.1	1.361370	0.240058
49	530.703659	530.4	0.303659	0.057251
51	494.093839	494.8	-0.706161	-0.142716
53	458.718401	460.3	-1.581599	-0.343602

55	424.650128	426.8	-2.149872	-0.503719
57	392.023241	394.4	-2.376759	-0.602627
59	360.728543	363.3	-2.571457	-0.707805
61	330.727256	333.3	-2.572744	-0.771900
63	302.117223	304.8	-2.682777	-0.880176
65	274.984740	277.5	-2.515260	-0.906400
67	249.501807	251.7	-2.198193	-0.873338
69	225.678695	227.3	-1.621305	-0.713289
71	203.490065	204.1	-0.609935	-0.298841
73	182.947468	182.4	0.547468	0.300147
75	163.907583	162.0	1.907583	1.177520
77	146.259349	142.9	3.359349	2.350839
79	129.909743	125.4	4.509743	3.596286
81	114.852425	109.4	5.452425	4.983935

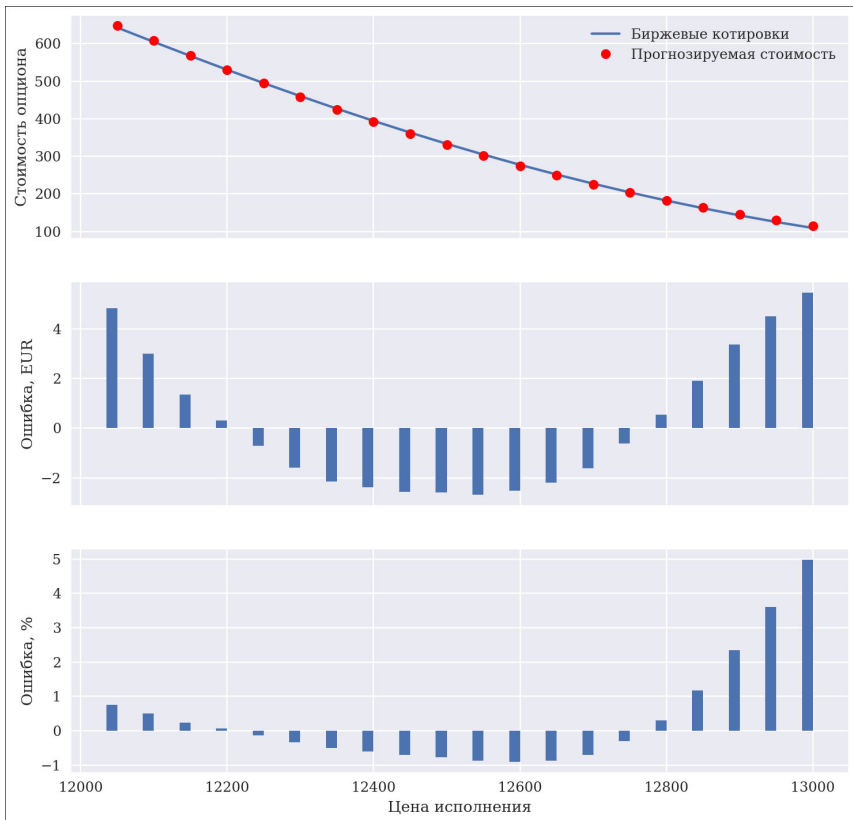
```
In [44]: round(option_selection['ERRORS_EUR'].mean(), 3) ❶
Out[44]: 0.184
```

```
In [45]: round(option_selection['ERRORS_%'].mean(), 3) ❷
Out[45]: 0.36
```

- ❶ Средняя ошибка оценки в евро.
- ❷ Средняя ошибка оценки в процентах.

Результаты оценки и ошибки оценки представлены на рис. 21.4.

```
In [46]: fig, (ax1, ax2, ax3) = plt.subplots(3, sharex=True,
                                             figsize=(10, 10))
strikes = option_selection['STRIKE_PRC'].values
ax1.plot(strikes, option_selection['CF_CLOSE'],
         label='Биржевые котировки')
ax1.plot(strikes, option_selection['MODEL'], 'ro',
         label='Прогнозируемая стоимость')
ax1.set_ylabel('Стоимость опциона')
ax1.legend(loc=0)
wi = 15
ax2.bar(strikes - wi / 2., option_selection['ERRORS_EUR'],
        width=wi)
ax2.set_ylabel('Ошибка, EUR')
ax3.bar(strikes - wi / 2., option_selection['ERRORS_%'],
        width=wi)
ax3.set_ylabel('Ошибка, %')
ax3.set_xlabel('Цена исполнения');
```



*Рис. 21.4. Сравнение прогнозируемых стоимостей и рыночных котировок после калибровки модели*



### Скорость калибровки

Полноценная калибровка модели оценки опционов по рыночным данным требует пересчета сотен или даже тысяч значений. Именно поэтому она обычно основывается на аналитических формулах ценообразования. В данном случае калибровка выполняется по методу Монте-Карло, которые намного требовательнее к производительности вычислительной системы, чем аналитические методы. Тем не менее процедура калибровки не длится “бесконечно долго” даже при запуске на обычном ноутбуке. Ее скорость можно существенно повысить за счет ряда методик, таких как векторизация и распараллеливание вычислений.

## Оценка портфеля

Получив в свое распоряжение откалиброванную модель, отражающую реалии финансовых рынков, которые представлены данными о котировках ликвидно торгуемых опционов, можно приступить к моделированию и оценке неторгуемых опционов. Идея заключается в том, что процедура калибровки равнозначна добавлению в модель корректной риск-нейтральной мартингальной меры благодаря настройке оптимальных параметров. На основе этой меры мы можем применить фундаментальную теорему ценообразования финансовых активов к остальным опционам, которые не участвовали в калибровке.

В этом разделе мы рассмотрим портфель американских пут-опционов индекса DAX. На биржах нет ликвидно торгуемых опционов такого типа. Для простоты будем считать, что американские пут-опционы имеют те же даты исполнения, что и европейские колл-опционы, использованные для калибровки модели. Также предполагается, что у них одинаковые страйк-цены.

## Моделирование опционных позиций

Прежде всего смоделируем рыночную среду для базового фактора риска — фондового индекса DAX — с оптимальными параметрами, полученными в результате калибровки.

```
In [47]: me_dax = market_environment('me_dax', pricing_date)
         me_dax.add_constant('initial_value', initial_value)
         me_dax.add_constant('final_date', pricing_date)
         me_dax.add_constant('currency', 'EUR')
```

```
In [48]: me_dax.add_constant('volatility', opt_local[0]) ❶
         me_dax.add_constant('lambda', opt_local[1]) ❶
         me_dax.add_constant('mu', opt_local[2]) ❶
         me_dax.add_constant('delta', opt_local[3]) ❶
```

```
In [49]: me_dax.add_constant('model', 'jd')
```

- ❶ Добавление оптимальных параметров, полученных в результате калибровки модели.

Далее определим опционные позиции и соответствующие им рыночные среды, сохранив их в двух разных словарях.

```
In [50]: payoff_func = 'np.maximum(strike - instrument_values, 0)'
```

```
In [51]: shared = market_environment('share', pricing_date)❶
```

```
shared.add_constant('maturity', maturity) ❶
shared.add_constant('currency', 'EUR') ❶
```

```
In [52]: option_positions = {}
option_environments = {}
for option in option_selection.index:
    option_environments[option] = market_environment(
        'am_put_%d' % option, pricing_date) ❷
    strike = option_selection['STRIKE_PRC'].loc[option] ❸
    option_environments[option].add_constant('strike',
                                             strike) ❸
    option_environments[option].add_environment(shared) ❹
    option_positions['am_put_%d' % strike] = \
        derivatives_position(
            'am_put_%d' % strike,
            quantity=np.random.randint(10, 50),
            underlying='dax_model',
            mar_env=option_environments[option],
            otype='American',
            payoff_func=payoff_func) ❺
```

- ❶ Общий объект `dx.market_environment`, который послужит базисом для рыночных сред всех опционов.
- ❷ Новый объект `dx.market_environment` для релевантного американского пут-опциона.
- ❸ Добавление параметра страйк-цены опциона.
- ❹ Добавление элементов из общего объекта `dx.market_environment` в объект рыночной среды конкретного опциона.
- ❺ Объект `dx.derivatives_position` с произвольным количеством опционов.

## Портфель опционов

Для оценки портфеля американских пут-опционов нам нужна отдельная рыночная среда. Она будет включать основные параметры для расчета стоимости позиций и сбора статистики.

```
In [53]: val_env = market_environment('val_env', pricing_date)
val_env.add_constant('starting_date', pricing_date)
val_env.add_constant('final_date', pricing_date) ❶
val_env.add_curve('discount_curve', csr)
```

```
val_env.add_constant('frequency', 'B')
val_env.add_constant('paths', 25000)
```

```
In [54]: underlyings = {'dax_model' : me_dax} ❷
```

```
In [55]: portfolio = derivatives_portfolio('portfolio',
                                          option_positions, val_env, underlyings) ❸
```

```
In [56]: %time results = portfolio.get_statistics(fixed_seed=True)
CPU times: user 1min 5s, sys: 2.91 s, total: 1min 8s
Wall time: 38.2 s
```

```
In [57]: results.round(1)
```

```
Out[57]:
```

	name	quant.	value	curr.	pos_value \
0	am_put_12050	33	151.6	EUR	5002.8
1	am_put_12100	38	161.5	EUR	6138.4
2	am_put_12150	20	171.3	EUR	3426.8
3	am_put_12200	12	183.9	EUR	2206.6
4	am_put_12250	37	197.4	EUR	7302.8
5	am_put_12300	37	212.3	EUR	7853.9
6	am_put_12350	36	228.4	EUR	8224.1
7	am_put_12400	16	244.3	EUR	3908.4
8	am_put_12450	17	262.7	EUR	4465.6
9	am_put_12500	16	283.4	EUR	4534.8
10	am_put_12550	38	305.3	EUR	11602.3
11	am_put_12600	10	330.4	EUR	3303.9
12	am_put_12650	38	355.5	EUR	13508.3
13	am_put_12700	40	384.2	EUR	15367.5
14	am_put_12750	13	413.5	EUR	5375.7
15	am_put_12800	49	445.0	EUR	21806.6
16	am_put_12850	30	477.4	EUR	14321.8
17	am_put_12900	33	510.3	EUR	16840.1
18	am_put_12950	40	544.4	EUR	21777.0
19	am_put_13000	35	582.3	EUR	20378.9

	pos_delta	pos_vega
0	-4.7	38206.9
1	-5.7	51365.2
2	-3.3	27894.5
3	-2.2	18479.7
4	-7.3	59423.5
5	-8.2	65911.9
6	-9.0	70969.4



7	-4.3	32871.4
8	-5.1	37451.2
9	-5.2	36158.2
10	-13.3	86869.9
11	-3.9	22144.5
12	-16.0	89124.8
13	-18.6	90871.2
14	-6.5	28626.0
15	-26.3	105287.3
16	-17.0	60757.2
17	-19.7	69163.6
18	-24.9	80472.3
19	-22.9	66522.6

```
In [58]: results[['pos_value', 'pos_delta', 'pos_vega']].sum().round(1)
Out[58]: pos_value      197346.2
         pos_delta      -224.0
         pos_vega      1138571.1
         dtype: float64
```

- ❶ Параметр `final_date` впоследствии сбрасывается к финальной дате исполнения для всех опционов портфеля.
- ❷ Все американские пут-опционы портфеля привязаны к одному и тому же фактору риска — фондовому индексу DAX.
- ❸ Создание объекта `dx.derivatives_portfolio`.

Расчет статистических показателей займет какое-то время, поскольку он выполняется по методу наименьших квадратов Монте-Карло, который оказывается особенно трудоемким в случае американских опционов. Так как мы имеем дело только с длинными позициями американских пут-опционов, рассматриваемый портфель характеризуется маленькой дельтой и большой вегой.

## Код Python

Ниже показан код Python, предназначенный для получения данных опционов фондового индекса DAX с помощью программного интерфейса Eikon.

```
In [1]: import eikon as ek ❶
        import pandas as pd
        import datetime as dt
        import configparser as cp
```

```
In [2]: cfg = cp.ConfigParser() ❷
        cfg.read('eikon.cfg') ❷
Out[2]: ['eikon.cfg']

In [3]: ek.set_app_id(cfg['eikon']['app_id']) ❷

In [4]: fields = ['CF_DATE', 'EXPIR_DATE', 'PUTCALLIND',
                  'STRIKE_PRC', 'CF_CLOSE', 'IMP_VOLT'] ❸

In [5]: dax = ek.get_data('0#GDAXN8*.EX', fields=fields)[0] ❹

In [6]: dax.info() ❹
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 115 entries, 0 to 114
Data columns (total 7 columns):
Instrument    115 non-null object
CF_DATE       115 non-null object
EXPIR_DATE    114 non-null object
PUTCALLIND    114 non-null object
STRIKE_PRC    114 non-null float64
CF_CLOSE      115 non-null float64
IMP_VOLT      114 non-null float64
dtypes: float64(3), object(4)
memory usage: 6.4+ KB
```

```
In [7]: dax['Instrument'] = dax['Instrument'].apply(
        lambda x: x.replace('/', '')) ❺
```

```
In [8]: dax.set_index('Instrument').head(10)
Out[8]:
```

	CF_DATE	EXPIR_DATE	PUTCALLIND \
<b>Instrument</b>			
.GDAXI	2018-04-27	None	None
GDAX105000G8.EX	2018-04-27	2018-07-20	CALL
GDAX105000S8.EX	2018-04-27	2018-07-20	PUT
GDAX108000G8.EX	2018-04-27	2018-07-20	CALL
GDAX108000S8.EX	2018-04-26	2018-07-20	PUT
GDAX110000G8.EX	2018-04-27	2018-07-20	CALL
GDAX110000S8.EX	2018-04-27	2018-07-20	PUT
GDAX111500G8.EX	2018-04-27	2018-07-20	CALL

GDAX111500S8.EX	2018-04-27	2018-07-20	PUT
GDAX112000G8.EX	2018-04-27	2018-07-20	CALL

	STRIKE_PRC	CF_CLOSE	IMP_VOLT
Instrument			
.GDAXI	NaN	12500.47	NaN
GDAX105000G8.EX	10500.0	2040.80	23.59
GDAX105000S8.EX	10500.0	32.00	23.59
GDAX108000G8.EX	10800.0	1752.40	22.02
GDAX108000S8.EX	10800.0	43.80	22.02
GDAX110000G8.EX	11000.0	1562.80	21.00
GDAX110000S8.EX	11000.0	54.50	21.00
GDAX111500G8.EX	11150.0	1422.50	20.24
GDAX111500S8.EX	11150.0	64.30	20.25
GDAX112000G8.EX	11200.0	1376.10	19.99

In [9]: `dax.to_csv('../source/tr_eikon_option_data.csv')` ⑥

- ① Импорт оболочечного пакета `eikon`.
- ② Считывание учетных данных для программного интерфейса `Eikon`.
- ③ Определение извлекаемых полей.
- ④ Получение данных опционов с экспирацией в июле 2018 года.
- ⑤ Замена символов косой черты ('/') в названиях инструментов.
- ⑥ Запись набора данных в CSV-файл.

## Резюме

В этой главе мы рассмотрели более крупный и реалистичный пример использования аналитического пакета `DX` для оценки портфеля неторгуемых американских опционов немецкого фондового индекса `DAX`. В любом приложении, связанном с анализом деривативов, приходится решать три основные задачи.

### *Получение данных*

Для моделирования и оценки деривативов требуются актуальные рыночные данные. В нашем случае мы использовали данные индекса `DAX` и котировки его опционов.

## Калибровка модели

Чтобы иметь возможность оценивать и хеджировать неторгуемые опционы рыночным способом, необходимо откалибровать параметры используемой модели (объекта моделирования) по соответствующим рыночным котировкам опционов (которые должны быть релевантны по срокам исполнения и страйк-ценам). Нами была выбрана модель прыжковой диффузии, которая в некоторых случаях достаточно хорошо описывает поведение фондовых индексов. Результаты калибровки оказались вполне удовлетворительными, несмотря на то что модель характеризуется только тремя степенями свободы ( $\lambda$  — интенсивность скачка,  $\mu$  — ожидаемый размер скачка,  $\sigma$  — вариативность размера скачка).

## Оценка портфеля

Собрав рыночные данные и получив откалиброванную модель, мы смоделировали портфель американских пут-опционов индекса DAX и рассчитали его основные статистические показатели (стоимости позиций, значения дельты и веги).

Этот пример призван продемонстрировать гибкость и функциональность пакета DX, который позволяет решать все основные задачи, связанные с анализом деривативов. Используемый подход и архитектура приложения во многом сопоставимы с эталонным случаем применения аналитической формулы Блэка — Шоулза — Мертона для европейских опционов. Как только объекты оценки определены, их можно использовать примерно так же, как и аналитическую формулу, даже несмотря на то, что для численных расчетов в них применяются вычислительно сложные и ресурсоемкие алгоритмы.

## Дополнительные ресурсы

Как и в предыдущих главах, хорошим справочным руководством по теме главы (особенно что касается калибровки модели оценки опционов) послужит книга Хилпиша [2].

1. Black, Fischer, and Myron Scholes. *The Pricing of Options and Corporate Liabilities* (1973, *Journal of Political Economy*, Vol. 81, No. 3, pp. 637–654).
2. Hilpisch, Yves. *Derivatives Analytics with Python* (2015, Wiley).
3. Merton, Robert. *Theory of Rational Option Pricing* (1973, *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141–183).

См. также список литературы в конце главы 20.



---

## Приложения



---

# Обработка значений даты и времени

Значения даты и времени играют важную роль в финансовых расчетах. В этом приложении мы рассмотрим различные способы работы с ними в коде Python. Конечно, данное описание не является исчерпывающим. Задача приложения — познакомить читателя с основными компонентами экосистемы Python, которые поддерживают обработку значений даты и времени.

## Python

Модуль `datetime` (<https://docs.python.org/3/library/datetime.html>) стандартной библиотеки Python позволяет выполнять большинство основных операций со значениями даты и времени.

```
In [1]: from pylab import mpl, plt
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

```
In [2]: import datetime as dt
```

```
In [3]: dt.datetime.now()❶
```

```
Out[3]: datetime.datetime(2018, 10, 19, 15, 17, 32, 164295)
```

```
In [4]: to = dt.datetime.today()❶
```

```
        to
```

```
Out[4]: datetime.datetime(2018, 10, 19, 15, 17, 32, 177092)
```

```
In [5]: type(to)
```

```
Out[5]: datetime.datetime
```

```
In [6]: dt.datetime.today().weekday()❷
```

```
Out[6]: 4
```



- ❶ Возвращает точную дату и системное время.
- ❷ Возвращает день недели в виде порядкового числа, где 0 соответствует понедельнику.

Объекты `datetime` можно создавать напрямую.

```
In [7]: d = dt.datetime(2020, 10, 31, 10, 5, 30, 500000) ❶  
d  
Out[7]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)
```

```
In [8]: str(d) ❷  
Out[8]: '2020-10-31 10:05:30.500000'
```

```
In [9]: print(d) ❸  
2020-10-31 10:05:30.500000
```

```
In [10]: d.year ❹  
Out[10]: 2020
```

```
In [11]: d.month ❺  
Out[11]: 10
```

```
In [12]: d.day ❻  
Out[12]: 31
```

```
In [13]: d.h ❼  
Out[13]: 10
```

- ❶ Пользовательский объект `datetime`.
- ❷ Строковое представление даты и времени.
- ❸ Вывод объекта на экран.
- ❹ Атрибуты `year` (год)...
- ❺ `...month` (месяц)...
- ❻ `...day` (день)...
- ❼ `...и hour` (час) объекта `datetime`.

Над объектом `datetime` можно выполнять различные преобразования.

```
In [14]: o = d.toordinal() ❶  
o
```

```
Out[14]: 737729
```

```
In [15]: dt.datetime.fromordinal(o) ❷  
Out[15]: datetime.datetime(2020, 10, 31, 0, 0)
```

```
In [16]: t = dt.datetime.time(d) ❸  
t  
Out[16]: datetime.time(10, 5, 30, 500000)
```

```
In [17]: type(t)  
Out[17]: datetime.time
```

```
In [18]: dd = dt.datetime.date(d) ❹  
dd  
Out[18]: datetime.date(2020, 10, 31)
```

```
In [19]: d.replace(second=0, microsecond=0) ❺  
Out[19]: datetime.datetime(2020, 10, 31, 10, 5)
```

- ❶ Преобразование в порядковое число.
- ❷ Преобразование из порядкового числа.
- ❸ Выделение компонента `time` (время).
- ❹ Выделение компонента `date` (дата).
- ❺ Установка заданных значений в 0.

При выполнении определенных арифметических операций с объектами `datetime` (например, вычисление разницы между двумя датами) результирующие значения могут иметь тип `timedelta`.

```
In [20]: td = d - dt.datetime.now() ❶  
td  
Out[20]: datetime.timedelta(days=742, seconds=67678,  
microseconds=169720)
```

```
In [21]: type(td) ❷  
Out[21]: datetime.timedelta
```

```
In [22]: td.days  
Out[22]: 742
```

```
In [23]: td.seconds
```

```
Out[23]: 67678
```

```
In [24]: td.microseconds
```

```
Out[24]: 169720
```

```
In [25]: td.total_seconds() ❸
```

```
Out[25]: 64176478.16972
```

❶ Объект разности между двумя объектами `datetime...`

❷ имеет тип `timedelta`.

❸ Разница между двумя датами в секундах.

Объект `datetime` можно преобразовывать в различные представления. Кроме того, можно создать такой объект из строки. Детали приведены в документации к модулю `datetime`. Рассмотрим несколько примеров.

```
In [26]: d.isoformat() ❶
```

```
Out[26]: '2020-10-31T10:05:30.500000'
```

```
In [27]: d.strftime('%A, %d. %B %Y %I:%M%p') ❷
```

```
Out[27]: 'Saturday, 31. October 2020 10:05AM'
```

```
In [28]: dt.datetime.strptime('2017-03-31', '%Y-%m-%d') ❸
```

```
Out[28]: datetime.datetime(2017, 3, 31, 0, 0)
```

```
In [29]: dt.datetime.strptime('30-4-16', '%d-%m-%y') ❸
```

```
Out[29]: datetime.datetime(2016, 4, 30, 0, 0)
```

```
In [30]: ds = str(d)
```

```
ds
```

```
Out[30]: '2020-10-31 10:05:30.500000'
```

```
In [31]: dt.datetime.strptime(ds, '%Y-%m-%d %H:%M:%S.%f') ❸
```

```
Out[31]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)
```

❶ Строковое представление в формате ISO.

❷ Представление даты в виде строкового шаблона.

❸ Создание объекта `datetime` на основе строкового шаблона.

Помимо функций `now()` и `today()` поддерживается также функция `utcnow()`, которая возвращает точную информацию о дате и времени в формате UTC (Coordinated Universal Time — всемирное координированное время),

ранее известном как GMT (Greenwich Mean Time — среднее время по Гринвичу). Оно может отличаться на несколько часов от местного часового пояса.

```
In [32]: dt.datetime.now()
```

```
Out[32]: datetime.datetime(2018, 10, 19, 15, 17, 32, 438889)
```

```
In [33]: dt.datetime.utcnow() ❶
```

```
Out[33]: datetime.datetime(2018, 10, 19, 13, 17, 32, 448897)
```

```
In [34]: dt.datetime.now() - dt.datetime.utcnow() ❷
```

```
Out[34]: datetime.timedelta(seconds=7199, microseconds=999995)
```

❶ Возвращает текущее время по UTC.

❷ Разница между местным временем и текущим временем по UTC.

В модуле `datetime` содержится общий класс часового пояса `tzinfo`, включающий методы `utcoffset()`, `dst()` и `tzname()`. На его основе можно, к примеру, создать классы, управляющие временем по UTC и CEST (Central European Summer Time — центральноевропейское летнее время).

```
In [35]: class UTC(dt.tzinfo):
```

```
    def utcoffset(self, d):
```

```
        return dt.timedelta(hours=0) ❶
```

```
    def dst(self, d):
```

```
        return dt.timedelta(hours=0) ❶
```

```
    def tzname(self, d):
```

```
        return 'UTC'
```

```
In [36]: u = dt.datetime.utcnow()
```

```
In [37]: u
```

```
Out[37]: datetime.datetime(2018, 10, 19, 13, 17, 32, 474585)
```

```
In [38]: u = u.replace(tzinfo=UTC()) ❷
```

```
In [39]: u
```

```
Out[39]: datetime.datetime(2018, 10, 19, 13, 17, 32, 474585,
```

```
            tzinfo=<__main__.UTC object at
```

```
            0x11c9a2320>)
```

```
In [40]: class CEST(dt.tzinfo):
```

```
    def utcoffset(self, d):
```

```
        return dt.timedelta(hours=2) ❸
```

```
def dst(self, d):
    return dt.timedelta(hours=1) ❸
def tzname(self, d):
    return 'CEST'
```

```
In [41]: c = u.astimezone(CEST()) ❹
c
```

```
Out[41]: datetime.datetime(2018, 10, 19, 15, 17, 32, 474585,
                           tzinfo=<__main__.CEST object at
                           0x11c9a2cc0>)
```

```
In [42]: c - c.dst() ❺
```

```
Out[42]: datetime.datetime(2018, 10, 19, 14, 17, 32, 474585,
                           tzinfo=<__main__.CEST object at
                           0x11c9a2cc0>)
```

- ❶ Нет смещения от UTC.
- ❷ Подключение объекта `dt.tzinfo` с помощью метода `replace()`.
- ❸ Смещение для стандартного и летнего (DST — Daylight Saving Time) времени по CEST.
- ❹ Переход из часового пояса UTC в часовой пояс CEST.
- ❺ Получение летнего времени для преобразованного объекта `datetime`.

В Python также имеется отдельный модуль `pytz`, реализующий поддержку основных часовых поясов.

```
In [43]: import pytz
```

```
In [44]: pytz.country_names['US'] ❶
Out[44]: 'United States'
```

```
In [45]: pytz.country_timezones['BE'] ❷
Out[45]: ['Europe/Brussels']
```

```
In [46]: pytz.common_timezones[-10:] ❸
Out[46]: ['Pacific/Wake',
          'Pacific/Wallis',
          'US/Alaska',
          'US/Arizona',
          'US/Central',
          'US/Eastern',
```

```
'US/Hawaii',  
'US/Mountain',  
'US/Pacific',  
'UTC']
```

- ❶ Название страны.
- ❷ Отдельный часовой пояс.
- ❸ Несколько часовых поясов.

Благодаря модулю `pytz` нет необходимости создавать пользовательские объекты `tzinfo`.

```
In [47]: u = dt.datetime.utcnow()
```

```
In [48]: u = u.replace(tzinfo=pytz.utc) ❶
```

```
In [49]: u
```

```
Out[49]: datetime.datetime(2018, 10, 19, 13, 17, 32, 611417,  
                           tzinfo=<UTC>)
```

```
In [50]: u.astimezone(pytz.timezone('CET')) ❷
```

```
Out[50]: datetime.datetime(2018, 10, 19, 15, 17, 32, 611417,  
                           tzinfo=<DstTzInfo 'CET'  
                           CEST+2:00:00 DST>)
```

```
In [51]: u.astimezone(pytz.timezone('GMT')) ❷
```

```
Out[51]: datetime.datetime(2018, 10, 19, 13, 17, 32, 611417,  
                           tzinfo=<StaticTzInfo 'GMT'>)
```

```
In [52]: u.astimezone(pytz.timezone('US/Central')) ❷
```

```
Out[52]: datetime.datetime(2018, 10, 19, 8, 17, 32, 611417,  
                           tzinfo=<DstTzInfo 'US/Central'  
                           CDT-1 day, 19:00:00 DST>)
```

- ❶ Определение часового пояса с помощью модуля `pytz`.
- ❷ Преобразование объекта `datetime` в разные часовые пояса.

## NumPy

Библиотека NumPy содержит собственные программные инструменты для работы со значениями даты и времени.

```
In [53]: import numpy as np
```

```
In [54]: nd = np.datetime64('2020-10-31') ❶  
nd
```

```
Out[54]: numpy.datetime64('2020-10-31')
```

```
In [55]: np.datetime_as_string(nd) ❶  
Out[55]: '2020-10-31'
```

```
In [56]: np.datetime_data(nd) ❷  
Out[56]: ('D', 1)
```

```
In [57]: d  
Out[57]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)
```

```
In [58]: nd = np.datetime64(d) ❸  
nd  
Out[58]: numpy.datetime64('2020-10-31T10:05:30.500000')
```

```
In [59]: nd.astype(dt.datetime) ❹  
Out[59]: datetime.datetime(2020, 10, 31, 10, 5, 30, 500000)
```

- ❶ Создание объекта даты из строки и получение строкового представления даты.
- ❷ Метаинформация об объекте даты (тип, размер).
- ❸ Создание объекта даты на основе объекта `datetime`.
- ❹ Преобразование в объект `datetime`.

Другой способ создания объекта даты заключается в предоставлении строкового шаблона, включающего, например, год, месяц и частоту повторения. Обратите внимание на то, что по умолчанию значение такого объекта задается равным первому дню месяца. Также можно создавать массивы дат на основе списков.

```
In [60]: nd = np.datetime64('2020-10', 'D')  
nd  
Out[60]: numpy.datetime64('2020-10-01')
```

```
In [61]: np.datetime64('2020-10') == np.datetime64('2020-10-01')  
Out[61]: True
```

```
In [62]: np.array(['2020-06-10', '2020-07-10', '2020-08-10'],
                  dtype='datetime64')
Out[62]: array(['2020-06-10', '2020-07-10', '2020-08-10'],
               dtype='datetime64[D'])
```

```
In [63]: np.array(['2020-06-10T12:00:00', '2020-07-10T12:00:00',
                  '2020-08-10T12:00:00'], dtype='datetime64[s]')
Out[63]: array(['2020-06-10T12:00:00', '2020-07-10T12:00:00',
               '2020-08-10T12:00:00'], dtype='datetime64[s'])
```

Функция `np.arange()` позволяет генерировать диапазоны дат. В ней можно указывать частоту повторения (например, дни, недели или секунды).

```
In [64]: np.arange('2020-01-01', '2020-01-04', dtype='datetime64') ❶
Out[64]: array(['2020-01-01', '2020-01-02', '2020-01-03'],
               dtype='datetime64[D'])
```

```
In [65]: np.arange('2020-01-01', '2020-10-01',
                  dtype='datetime64[M']) ❷
Out[65]: array(['2020-01', '2020-02', '2020-03', '2020-04',
               '2020-05', '2020-06', '2020-07', '2020-08',
               '2020-09'], dtype='datetime64[M'])
```

```
In [66]: np.arange('2020-01-01', '2020-10-01',
                  dtype='datetime64[W]')[1:10] ❸
Out[66]: array(['2019-12-26', '2020-01-02', '2020-01-09',
               '2020-01-16', '2020-01-23', '2020-01-30',
               '2020-02-06', '2020-02-13', '2020-02-20',
               '2020-02-27'], dtype='datetime64[W'])
```

```
In [67]: dtl = np.arange('2020-01-01T00:00:00',
                        '2020-01-02T00:00:00',
                        dtype='datetime64[h]') ❹
        dtl[1:10]
Out[67]: array(['2020-01-01T00', '2020-01-01T01', '2020-01-01T02',
               '2020-01-01T03', '2020-01-01T04', '2020-01-01T05',
               '2020-01-01T06', '2020-01-01T07', '2020-01-01T08',
               '2020-01-01T09'], dtype='datetime64[h'])
```

```
In [68]: np.arange('2020-01-01T00:00:00', '2020-01-02T00:00:00',
                  dtype='datetime64[s]')[1:10] ❺
Out[68]: array(['2020-01-01T00:00:00', '2020-01-01T00:00:01',
               '2020-01-01T00:00:02', '2020-01-01T00:00:03',
               '2020-01-01T00:00:04', '2020-01-01T00:00:05',
```



```
'2020-01-01T00:00:06', '2020-01-01T00:00:07',
'2020-01-01T00:00:08', '2020-01-01T00:00:09'],
dtype='datetime64[s]')
```

```
In [69]: np.arange('2020-01-01T00:00:00', '2020-01-02T00:00:00',
dtype='datetime64[ms]')[10] ⑥
```

```
Out[69]: array(['2020-01-01T00:00:00.000', '2020-01-01T00:00:00.001',
'2020-01-01T00:00:00.002', '2020-01-01T00:00:00.003',
'2020-01-01T00:00:00.004', '2020-01-01T00:00:00.005',
'2020-01-01T00:00:00.006', '2020-01-01T00:00:00.007',
'2020-01-01T00:00:00.008', '2020-01-01T00:00:00.009'],
dtype='datetime64[ms]')
```

- ① Ежедневное повторение.
- ② Ежемесячное повторение.
- ③ Еженедельное повторение.
- ④ Повторение через час.
- ⑤ Повторение через секунду.
- ⑥ Повторение через миллисекунду.

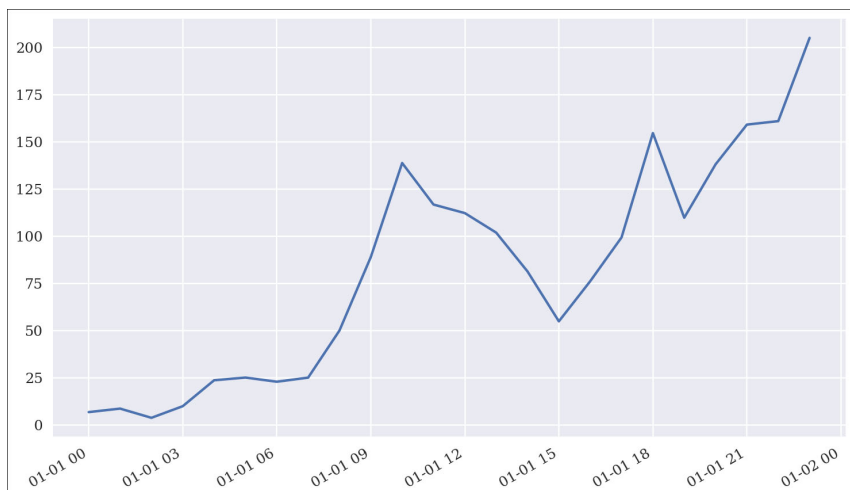
Построение графиков временных рядов иногда оказывается непростой задачей. В библиотеке `matplotlib` реализована поддержка стандартных объектов `datetime`, поэтому обычно достаточно преобразовать объект `datetime64` библиотеки NumPy в объект `datetime` стандартной библиотеки Python, как показано в следующем примере (рис. A.1).

```
In [70]: import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [71]: np.random.seed(3000)
rnd = np.random.standard_normal(len(dtl)).cumsum() ** 2
```

```
In [72]: fig = plt.figure(figsize=(10, 6))
plt.plot(dtl.astype(dt.datetime), rnd) ①
fig.autofmt_xdate(); ②
```

- ① Значения  $x$  представляются объектами `datetime`.
- ② Автоматическое форматирование подписей оси X в виде значений даты/времени.



*Рис. А.1.1. График с автоматически отформатированными подписями оси X, которые представлены объектами `datetime`*

## pandas

Библиотека `pandas` изначально разрабатывалась в расчете на поддержку временных рядов (см. официальную документацию по адресу [http://bit.ly/timeseries\\_doc](http://bit.ly/timeseries_doc)), поэтому она включает классы, предназначенные для эффективной обработки значений даты и времени, например `DatetimeIndex`.

В библиотеке `pandas` также имеется объект `Timestamp`, который служит альтернативой объектам `datetime` и `datetime64`.

```
In [73]: import pandas as pd
```

```
In [74]: ts = pd.Timestamp('2020-06-30') ❶
         ts
```

```
Out[74]: Timestamp('2020-06-30 00:00:00')
```

```
In [75]: d = ts.to_pydatetime() ❷
         d
```

```
Out[75]: datetime.datetime(2020, 6, 30, 0, 0)
```

```
In [76]: pd.Timestamp(d) ❸
Out[76]: Timestamp('2020-06-30 00:00:00')
```

```
In [77]: pd.Timestamp(nd) ❹  
Out[77]: Timestamp('2020-10-01 00:00:00')
```

- ❶ Создание объекта `Timestamp` из строки.
- ❷ Создание объекта `datetime` из объекта `Timestamp`.
- ❸ Создание объекта `Timestamp` из объекта `datetime`.
- ❹ Создание объекта `Timestamp` из объекта `datetime64`.

Вышеупомянутый класс `DatetimeIndex` ([http://bit.ly/datetimeindex\\_doc](http://bit.ly/datetimeindex_doc)) представляет собой коллекцию объектов `Timestamp`, снабженную рядом полезных методов. Объект `DatetimeIndex` можно создать с помощью функции `pd.date_range()` ([http://bit.ly/date\\_range\\_doc](http://bit.ly/date_range_doc); см. главу 5), которая позволяет легко строить индексы временных рядов. Ниже приведено несколько примеров преобразований дат.

```
In [78]: dti = pd.date_range('2020/01/01', freq='M', periods=12) ❶  
        dti  
Out[78]: DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31',  
                        '2020-04-30', '2020-05-31', '2020-06-30',  
                        '2020-07-31', '2020-08-31', '2020-09-30',  
                        '2020-10-31', '2020-11-30', '2020-12-31'],  
                        dtype='datetime64[ns]', freq='M')
```

```
In [79]: dti[6]  
Out[79]: Timestamp('2020-07-31 00:00:00', freq='M')
```

```
In [80]: pdi = dti.to_pydatetime() ❷  
        pdi  
Out[80]: array([datetime.datetime(2020, 1, 31, 0, 0),  
               datetime.datetime(2020, 2, 29, 0, 0),  
               datetime.datetime(2020, 3, 31, 0, 0),  
               datetime.datetime(2020, 4, 30, 0, 0),  
               datetime.datetime(2020, 5, 31, 0, 0),  
               datetime.datetime(2020, 6, 30, 0, 0),  
               datetime.datetime(2020, 7, 31, 0, 0),  
               datetime.datetime(2020, 8, 31, 0, 0),  
               datetime.datetime(2020, 9, 30, 0, 0),  
               datetime.datetime(2020, 10, 31, 0, 0),  
               datetime.datetime(2020, 11, 30, 0, 0),  
               datetime.datetime(2020, 12, 31, 0, 0)],  
               dtype=object)
```

```
In [81]: pd.DatetimeIndex(pdi) ❸
Out[81]: DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31',
                        '2020-04-30', '2020-05-31', '2020-06-30',
                        '2020-07-31', '2020-08-31', '2020-09-30',
                        '2020-10-31', '2020-11-30', '2020-12-31'],
                        dtype='datetime64[ns]', freq=None)
```

```
In [82]: pd.DatetimeIndex(dtl) ❹
Out[82]: DatetimeIndex(['2020-01-01 00:00:00', '2020-01-01 01:00:00',
                        '2020-01-01 02:00:00', '2020-01-01 03:00:00',
                        '2020-01-01 04:00:00', '2020-01-01 05:00:00',
                        '2020-01-01 06:00:00', '2020-01-01 07:00:00',
                        '2020-01-01 08:00:00', '2020-01-01 09:00:00',
                        '2020-01-01 10:00:00', '2020-01-01 11:00:00',
                        '2020-01-01 12:00:00', '2020-01-01 13:00:00',
                        '2020-01-01 14:00:00', '2020-01-01 15:00:00',
                        '2020-01-01 16:00:00', '2020-01-01 17:00:00',
                        '2020-01-01 18:00:00', '2020-01-01 19:00:00',
                        '2020-01-01 20:00:00', '2020-01-01 21:00:00',
                        '2020-01-01 22:00:00', '2020-01-01 23:00:00'],
                        dtype='datetime64[ns]', freq=None)
```

- ❶ Объект `DatetimeIndex`, содержащий 12 периодов с ежемесячной частотой повторения.
- ❷ Преобразование объекта `DatetimeIndex` в массив `ndarray`, содержащий объекты `datetime`.
- ❸ Создание объекта `DatetimeIndex` на основе массива `ndarray`, содержащего объекты `datetime`.
- ❹ Создание объекта `DatetimeIndex` на основе массива `ndarray`, содержащего объекты `datetime64`.

Библиотека `pandas` позволяет строить наглядные графики временных зависимостей (рис. А.2). Дополнительные примеры были приведены в главе 8.

```
In [83]: rnd = np.random.standard_normal(len(dti)).cumsum() ** 2
```

```
In [84]: df = pd.DataFrame(rnd, columns=['Данные'], index=dti)
```

```
In [85]: df.plot(figsize=(10, 6));
```



*Рис. А.2. Построенный библиотекой pandas график с автоматически отформатированными подписями оси X, которые представлены объектами Timestamp*

Библиотека pandas хорошо интегрируется с модулем pytz при работе с часовыми поясами.

```
In [86]: pd.date_range('2020/01/01', freq='M', periods=12,
                      tz=pytz.timezone('CET'))
Out[86]: DatetimeIndex(['2020-01-31 00:00:00+01:00',
                        '2020-02-29 00:00:00+01:00', '2020-03-31 00:00:00+02:00',
                        '2020-04-30 00:00:00+02:00', '2020-05-31 00:00:00+02:00',
                        '2020-06-30 00:00:00+02:00', '2020-07-31 00:00:00+02:00',
                        '2020-08-31 00:00:00+02:00', '2020-09-30 00:00:00+02:00',
                        '2020-10-31 00:00:00+01:00', '2020-11-30 00:00:00+01:00',
                        '2020-12-31 00:00:00+01:00'], dtype='datetime64[ns, CET]',
                      freq='M')
```

```
In [87]: dti = pd.date_range('2020/01/01', freq='M', periods=12,
                             tz='US/Eastern')
dti
Out[87]: DatetimeIndex(['2020-01-31 00:00:00-05:00',
                        '2020-02-29 00:00:00-05:00', '2020-03-31 00:00:00-04:00',
                        '2020-04-30 00:00:00-04:00', '2020-05-31 00:00:00-04:00',
                        '2020-06-30 00:00:00-04:00', '2020-07-31 00:00:00-04:00',
                        '2020-08-31 00:00:00-04:00', '2020-09-30 00:00:00-04:00',
```

```
'2020-10-31 00:00:00-04:00', '2020-11-30 00:00:00-05:00',  
'2020-12-31 00:00:00-05:00'],  
dtype='datetime64[ns, US/Eastern]', freq='M')
```

```
In [88]: dti.tz_convert('GMT')
```

```
Out[88]: DatetimeIndex(['2020-01-31 05:00:00+00:00',  
                        '2020-02-29 05:00:00+00:00', '2020-03-31 04:00:00+00:00',  
                        '2020-04-30 04:00:00+00:00', '2020-05-31 04:00:00+00:00',  
                        '2020-06-30 04:00:00+00:00', '2020-07-31 04:00:00+00:00',  
                        '2020-08-31 04:00:00+00:00', '2020-09-30 04:00:00+00:00',  
                        '2020-10-31 04:00:00+00:00', '2020-11-30 05:00:00+00:00',  
                        '2020-12-31 05:00:00+00:00'], dtype='datetime64[ns, GMT]',  
                        freq='M')
```



# Класс опционов в модели Блэка — Шоулза — Мертона

## Определение класса

Ниже приведено определение класса для европейского колл-опциона в модели Блэка — Шоулза — Мертона. Подход на основе класса можно считать альтернативой реализации с помощью функций, которая рассматривалась в главе 12.

```
#
# Оценка европейских колл-опционов в модели
# Блэка — Шоулза — Мертона (BSM), включая функции
# вычисления веги и ожидаемой волатильности.
# Реализация на основе класса.
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
from math import log, sqrt, exp
from scipy import stats

class bsm_call_option(object):
    ''' Класс оценки европейских колл-опционов
        в модели Блэка — Шоулза — Мертона.

    Атрибуты
    =====
    S0: float
        Начальный уровень акции/индекса
    K: float
        Страйк-цена
    T: float
        Срок исполнения (в долях года)
    r: float
        Постоянная безрисковая краткосрочная ставка
```



***sigma: float***

*Коэффициент волатильности компонента диффузии*

***Методы***

***=====***

***value:***

*Возвращает текущую стоимость колл-опциона*

***vega:***

*Возвращает вегу колл-опциона*

***imp\_vol:***

*Возвращает ожидаемую волатильность*

***'''***

```
def __init__(self, S0, K, T, r, sigma):
```

```
    self.S0 = float(S0)
```

```
    self.K = K
```

```
    self.T = T
```

```
    self.r = r
```

```
    self.sigma = sigma
```

```
def value(self):
```

```
    ''' Возвращает стоимость опциона
```

```
    '''
```

```
    d1 = ((log(self.S0 / self.K) +  
           (self.r + 0.5 * self.sigma ** 2) * self.T) /  
           (self.sigma * sqrt(self.T)))
```

```
    d2 = ((log(self.S0 / self.K) +  
           (self.r - 0.5 * self.sigma ** 2) * self.T) /  
           (self.sigma * sqrt(self.T)))
```

```
    value = (self.S0 * stats.norm.cdf(d1, 0.0, 1.0) -  
            self.K * exp(-self.r * self.T) *  
            stats.norm.cdf(d2, 0.0, 1.0))
```

```
    return value
```

```
def vega(self):
```

```
    ''' Возвращает вегу опциона
```

```
    '''
```

```
    d1 = ((log(self.S0 / self.K) +  
           (self.r + 0.5 * self.sigma ** 2) * self.T) /  
           (self.sigma * sqrt(self.T)))
```

```
    vega = self.S0 * stats.norm.pdf(d1, 0.0, 1.0) * sqrt(self.T)  
    return vega
```

```
def imp_vol(self, C0, sigma_est=0.2, it=100):
    ''' Возвращает ожидаемую волатильность
        для заданной цены опциона
    '''
    option = bsm_call_option(self.S0, self.K, self.T, self.r,
                             sigma_est)
    for i in range(it):
        option.sigma -= (option.value() - C0) / option.vega()
    return option.sigma
```

## Пример использования

Созданный класс можно применить в отдельном интерактивном сеансе Jupyter Notebook.

```
In [1]: from bsm_option_class import *

In [2]: o = bsm_call_option(100., 105., 1.0, 0.05, 0.2)
         type(o)
Out[2]: bsm_option_class.bsm_call_option

In [3]: value = o.value()
         value
Out[3]: 8.021352235143176

In [4]: o.vega()
Out[4]: 39.67052380842653

In [5]: o.imp_vol(C0=value)
Out[5]: 0.2
```

С помощью данного класса можно, например, визуализировать стоимость и вегу опциона для разных страйк-цен и сроков исполнения. В конце концов, это одно из главных преимуществ наличия аналитической формулы оценки опциона. В следующем коде Python генерируются статистические показатели опциона для разных комбинаций страйк-цен и сроков исполнения.

```
In [6]: import numpy as np
         maturities = np.linspace(0.05, 2.0, 20)
         strikes = np.linspace(80, 120, 20)
         T, K = np.meshgrid(strikes, maturities)
         C = np.zeros_like(K)
         V = np.zeros_like(C)
```

```

for t in enumerate(maturities):
    for k in enumerate(strikes):
        o.T = t[1]
        o.K = k[1]
        C[t[0], k[0]] = o.value()
        V[t[0], k[0]] = o.vega()

```

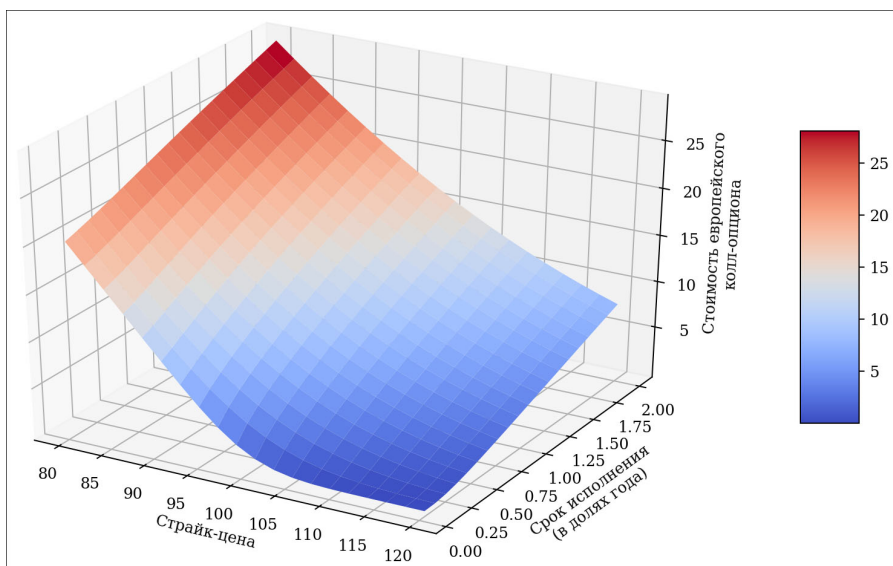
Ценовая поверхность европейского колл-опциона показана на рис. Б.1.

```

In [7]: from pylab import cm, mpl, plt
        from mpl_toolkits.mplot3d import Axes3D
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline

In [8]: fig = plt.figure(figsize=(12, 7))
        ax = fig.gca(projection='3d')
        surf = ax.plot_surface(T, K, C, rstride=1, cstride=1,
                               cmap=cm.coolwarm, linewidth=0.5,
                               antialiased=True)
        ax.set_xlabel('Страйк-цена')
        ax.set_ylabel('\nСрок исполнения\n(в долях года)')
        ax.set_zlabel('Стоимость европейского\nколл-опциона')
        fig.colorbar(surf, shrink=0.5, aspect=5);

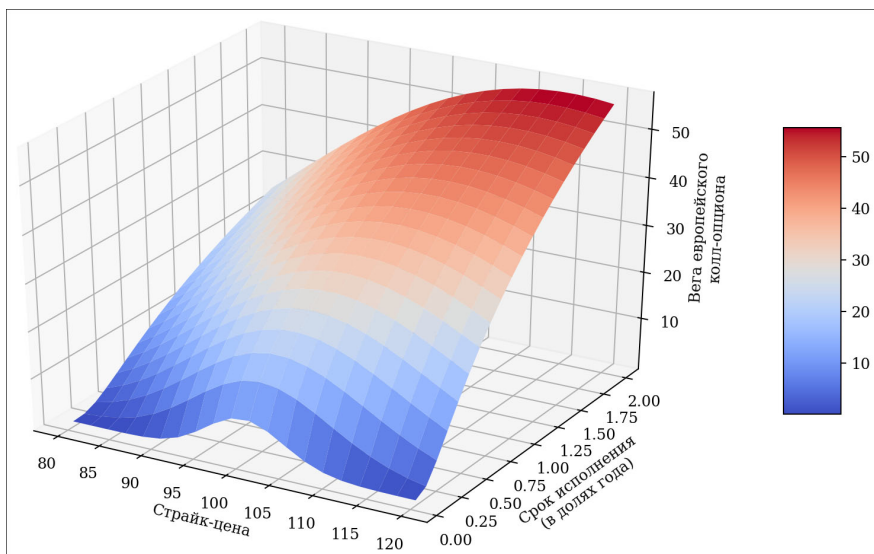
```



*Рис. Б.1. Ценовая поверхность европейского колл-опциона*

Поверхность значений вего европейского колл-опциона представлена на рис. Б.2.

```
In [9]: fig = plt.figure(figsize=(12, 7))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(T, K, V, rstride=1, cstride=1,
                      cmap=cm.coolwarm, linewidth=0.5,
                      antialiased=True)
ax.set_xlabel('Страйк-цена')
ax.set_ylabel('\nСрок исполнения\n(в долях года)')
ax.set_zlabel('Вега европейского\nколл-опциона')
fig.colorbar(surf, shrink=0.5, aspect=5);
```



**Рис. Б.2.** Поверхность значений вего европейского колл-опциона



---

# Предметный указатель

## B

BeautifulSoup 56  
Blosc 317  
BSM 41

## C

CFD 545  
conda 65  
CSV 256, 288, 299, 305  
Cufflinks 215, 243  
CVA 456  
Cython 334, 338

## D

D3.js 243  
DAX 739  
DNN 530, 596  
Docker 76  
Dockerfile 77  
DX Analytics 648

## E

Eikon 53, 259  
EMH 571  
EWMA 367  
Excel 306

## F

FIFO 287  
FXCM 545, 619  
fxcmpy 546, 619

## G

GMT 769  
GNB 525

## H

HDF5 308, 320

## I

IDE 32  
IPython 32, 69

## J

Jupyter 32  
Notebook 84

## K

KDE 277

## L

LSM 448, 691, 705

## M

matplotlib 34, 69  
MCMC 505  
Miniconda 65  
mkl 68  
MSE 748

## N

Notebook 32  
Numba 337  
numexpr 50  
NumPy 33, 121, 197, 295, 336  
    дата и время 771  
    установка 68

## O

OHLC 249  
OpenSSL 83

## **P**

pandas 34, 70, 153, 255  
временные ряды 775  
PyMC3 503  
PyTables 34, 70, 308

## **R**

RIC 54, 259  
RSA 83  
RSI 251  
RWH 571

## **S**

S&P 500 274  
Scikit-learn 34, 57, 70, 518  
нейронные сети 530, 596  
SciPy 34, 70  
SMA 271, 549, 562  
SQLite3 292, 300  
SVM 58, 537  
SymPy 401

## **T**

TensorFlow 57, 519, 531, 599  
TsTables 324

## **U**

Unicode 104  
UTC 768

## **V**

VaR 451, 631  
VIX 274

## **Z**

ZCB 656  
ZeroMQ 638

## **A**

Абсолютная разность 263  
Абсолютная цена 517  
Автоматизированная торговля 607  
Агрегирование 194, 205  
Азиатский опцион 699  
Актив 353  
Алгоритм 340  
итеративный 346  
рекурсивный 345  
Алгоритмическая торговля 543  
Американский опцион 448, 691, 705  
Анализ рисков 628  
Аппроксимация 374  
Арбитраж 651  
Аск 552  
Атрибут 191  
закрытый 204

## **Б**

База данных 292, 300  
HDF5 308, 320  
Байесовская регрессия 503  
Байесовская статистика 502  
Безрисковый актив , 353  
Бермудский опцион 448, 705  
Бета-коэффициент 467  
Бид 552  
Биномиальное дерево 353  
Биржевая заявка 557  
Биржевая торговля 614  
Большие данные 39  
Броуновское движение 467  
Булево значение 98

## **В**

Валютный курс 622  
Вега 696  
Вектор 122  
Векторизация 145, 336, 441, 491  
Векторизованное тестирование 561, 566, 620

Вероятность дефолта 456  
Вещественное число 95  
Виртуальное окружение 72  
Внешнее слияние 184  
Возврат к среднему 424  
Волатильность 490  
    ожидаемая 741  
    стохастическая 430  
Временное окно 269  
Временной ряд 161, 255, 324  
Выпуклое программирование 391, 748  
Высокочастотные данные 279

## Г

Генератор случайных чисел 339, 411  
Геометрическое броуновское движение 360, 418, 421, 673  
Гипотеза  
    случайного блуждания 571  
    эффективного рынка 467, 571  
Гистограмма 234, 247  
Глобальная оптимизация 392, 750  
Глубокое обучение 530  
Граница эффективности 492, 496  
График 168  
    двухмерный 216  
    легенда 225  
Греки 695  
Группирование 172

## Д

Дата и время 765  
Декоратор 346  
Дельта 695  
Денежность 744  
Дерево решений 528  
Дериватив  
    оценка 691  
    портфель 722  
Деривативная позиция 718  
Десериализация 285  
Дефолт 456

Децимация 266  
Диаграмма 168  
    ОНLC 249  
    интерактивная 243  
    размаха 236  
    рассеяния 231, 277, 510  
    тип 247  
    трехмерная 239  
    финансовая 248  
Диверсификация 486  
Динамическая компиляция 337  
Дисконтирование  
    риск-нейтральное 653  
Дисконтная облигация 656  
Дискретизация по методу Эйлера 435  
Дисперсия  
    ожидаемой доходности портфеля 490  
    уменьшение 438  
Дифференцирование 406  
Диффузия  
    по закону квадратного корня 424, 666, 684  
    прыжковая 435, 665  
Доходность  
    логарифмическая 265, 276  
    простая 264  
Дроплет 82

## Е

Европейский опцион 41, 442, 691, 696

## З

Зависимое наблюдение 375  
Запрос 302  
Зашумленные данные 381  
Заявка 557

## И

Импорт данных 256  
Индекс относительной силы 251  
Индексация 110  
Индикатор 268



Инкапсуляция 194, 204  
Интегрирование 398, 404  
    численное 400  
Интегрированная среда разработки 32  
Интерполяция 386  
Искусственный интеллект 57  
Исторические данные 479  
Итератор 210

## К

Калибровка модели 743, 748  
Квадратура 400  
Квантиль 474  
Класс 191, 201  
    array 124  
    DataFrame 154, 199, 299  
    EArray 321  
    ndarray 127, 197  
    Series 171  
    Symbol 401  
    Timestamp 775  
Классификация 523, 586  
Кластеризация 520, 581  
Ключ шифрования 83  
Ковариационная матрица 488, 489  
Компиляция 49  
    динамическая 337  
    статическая 338  
Конкатенация 179  
Контейнер 64  
    Docker 76  
Контракт на разницу цен 545  
Корреляционный анализ 274  
Корреляционный коэффициент 431  
Корреляция 278  
Кортеж 109  
Коэффициент  
    асимметрии 475  
    левериджа 616  
    Шарпа 491  
    экссесса 475  
Краткосрочная ставка 656  
Кредитная стоимость под риском 456

Критерий Келли 608  
Кубический сплайн 389

## Л

Леверидж 431, 616  
    оптимальный 626  
Левое соединение 181  
Линейная регрессия 575  
Линии Боллинджера 250  
Линия рынка капиталов 498  
Логарифмическая доходность 265, 276,  
    467  
Логистическая регрессия 527  
Логическая операция 100  
Логнормальное распределение 419, 468  
Локальная оптимизация 394, 751  
Лямбда-функция 115

## М

Марковская цепь 505  
Марковский момент 448, 706  
Марковское свойство 421  
Мартингал 442, 650  
Массив 122  
    EArray 321  
    PyTables 319  
    булев 139  
    изменение размера 136  
    инициализация 131  
    метаинформация 134  
    многомерный 130  
    разуплотнение 138  
    склеивание 137  
    структурированный 143  
    трансляция 146  
    форма 135  
Матрица 122  
    Холецкого 431  
Машинное обучение 518, 619  
Менеджер пакетов 64  
Мера риска 451, 456  
Метка 519

Метод 191  
  k-средних 520, 581  
  Гаусса 400  
  моментов 440  
  Монте-Карло 42, 349, 359, 400, 417, 691, 705  
    с марковскими цепями 505  
  наименьших квадратов 277, 448, 575, 691, 705  
  опорных векторов 58, 537  
  Ромберга 400  
  симметричных выборок 439  
  Симпсона 400  
  трапеций 400  
  Эйлера 360, 425  
Минимизация 375  
Многомерная зависимость 383  
Множество 117  
Модель  
  Блэка — Шоулза — Мертона 41, 417, 422, 467, 781  
  данных 206  
  Кокса — Росса — Рубинштейна 353  
  оценки опционов 353  
  финансового рынка 651  
  ценообразования капитальных активов 467  
Модуль  
  array 124  
  copy 124  
  csv 291  
  datetime 108, 765  
  decimal 97  
  Expr 322  
  keyword 98  
  math 31  
  matplotlib 215  
  multiprocessing 344, 365  
  numpy.random 410  
  pickle 285  
  plotly 215, 243  
  pytz 770  
  re 107  
  tables 308

Момент остановки 448, 706

Мониторинг 638

## Н

Наивный байесовский классификатор 525

Наследование 194

Научный стек 33

Независимое наблюдение 375

Нейронная сеть 530, 596

Нормализация 265

Нормализующая константа 502

Нормальное распределение 415, 466

## О

Облачный экземпляр 64, 82, 637

Образ Docker 76

Обучение

  без учителя 519

  с учителем 522

Объект 191

  неизменяемый 110

Объектно-ориентированное программирование (ООП) 191

Ожидаемая доходность 489

Ожидаемая полезность 395

Операционный риск 637

Оптимизация

  глобальная 392, 750

  локальная 394, 751

  условная 395

Опцион

  азиатский 699

  американский 448, 691, 705

  бермудский 448, 705

  вега 696

  дельта 695

  европейский 41, 442, 691, 696

  премия 650

  ценовая поверхность 784

Относительная доходность 517

Оценка рисков 451

Ошибка выборки 421

## П

Параллельные вычисления 343, 365  
Переобучение 570  
Полезность 395  
Полиморфизм 194  
Полный рынок 653  
Поправка на кредитный риск 456  
Портфель 717  
    ожидаемая доходность 489  
    оптимальный 493  
    оценка 723, 755  
Портфельная теория Марковица 466, 486  
Правдоподобие 502  
Премия  
    досрочного исполнения 450  
    за риск 650  
Продленная стоимость 448, 706  
Производительность 333  
Прореживание 266  
Просадка 628  
Простое число 340  
Протоколирование 638  
Процентное изменение 264  
Процент попаданий 623  
Прыжковая диффузия 435, 665, 679

## Р

Распределение  
    логнормальное 419  
    Пуассона 415  
    хи-квадрат 415  
Регрессионный анализ 277  
Регрессия 375  
    Байесовская 503  
    линейная 376, 575  
    многомерных зависимостей 383  
Регулярное выражение 107  
Редактор кода 82  
Рекурсия 345, 366  
Релевантный рынок 722  
Реляционная база данных 300

Риск 451, 628  
Риск-нейтральное дисконтирование 653  
Рисковый актив 353  
Рыночная среда 658, 669

## С

Свечные данные 550  
Сериализация 285  
Сжатая таблица 317  
Символьные вычисления 401  
Символьный идентификатор Reuters 259  
Скользкая статистика 268  
Скользкое среднее 271, 549, 562  
    экспоненциально взвешенное 367  
Скрытый слой 530  
Слияние 183  
Словарь 116  
    методы 117  
Случайная переменная 417  
Случайное блуждание 513, 571  
Случайное число 410, 666  
Смесь гауссиан 522  
Смещенный прогноз 268  
Соединение 181  
Специальный метод  
    \_\_abs\_\_() 207  
    \_\_add\_\_() 208  
    \_\_bool\_\_() 207  
    \_\_getitem\_\_() 209  
    \_\_init\_\_() 202, 206  
    \_\_iter\_\_() 210  
    \_\_len\_\_() 209  
    \_\_mul\_\_() 208  
    \_\_repr\_\_() 207  
    \_\_sizeof\_\_() 196  
Списковое включение 114  
Список 110, 196  
    методы 112  
    преобразование в массив 122  
Сравнение 99  
Среднеквадратическая ошибка 378, 748  
Срез 112

Стандартная библиотека 31  
Статистика 260  
Статистический анализ 465  
Статическая компиляция 338  
Стоимость под риском 451, 631  
    кредитная 456  
Стохастическая волатильность 430  
Стохастический процесс 421  
Страйк-цена 442  
Строка 102  
    вывод 104  
    замена 105  
    методы 103  
Структура данных 109

## Т

Таблица 317  
Теорема  
    Байеса 502  
    ценообразования финансовых активов 442, 650  
Терминал 82  
Технический анализ 268, 271  
Тиковые данные 279, 548  
Тип данных 94  
    bool 98  
    dict 116  
    float 95  
    int 94, 195  
    iterator 116  
    list 110, 196  
    range 113  
    set 117  
    str 102  
    tuple 109  
Торговая стратегия 561, 564  
Точность 96  
Трассировочный график 507  
Трехмерная поверхность 241

## У

Универсальная функция 129  
Управление капиталом 608  
Управляющая конструкция 112  
Условная оптимизация 395  
Условные требования 441  
Утяжеленный хвост 483

## Ф

Финансовые данные 256  
Финансовый анализ 39  
Финансовый инструмент 201  
Флеш-трейдинг 38  
Формула Байеса 502  
Функциональное программирование 114

Функция  
    deepcopy() 124  
    dir() 94  
    dump() 286  
    filter() 115  
    fsolve() 499  
    load() 286  
    map() 115  
    minimize() 493  
    pd.read\_csv() 256  
    plt.axis() 220  
    plt.boxplot() 236  
    plt.hist() 234  
    plt.legend() 225  
    plt.plot() 217  
    plt.plot\_surface() 241  
    plt.scatter() 232  
    plt.setp() 237  
    plt.subplot() 229  
    plt.subplots() 228  
    plt.title() 221  
    plt.xlabel() 221  
    plt.xlim() 220  
    plt.ylabel() 221  
    plt.ylim() 220  
    print() 104

rand() 410  
scatter\_matrix() 277  
type() 94  
базисная 375  
обратного вызова 556  
объявление 114  
рекурсивная 345, 366  
универсальная 129

## Ц

Целое число 94, 195  
Цена исполнения 442  
Цикл 334  
    for 112  
    while 114

## Ч

Часовой пояс 769  
Частотное распределение 362

Численное интегрирование 400

## Число

округление 97  
пи 349  
простое 340  
случайное 410, 666  
с плавающей точкой 95  
точность 96  
Фибоначчи 345  
целое 94, 195

## Э

Экземпляр класса 191

## Я

Ядерная оценка плотности 277

У новому виданні книги розробники та фінансові аналітики дізнаються, як застосовувати різні інструменти Python для створення фінансових додатків і систем алгоритмічної торгівлі. Всі приклади книги написані на Python 3 і доступні у вигляді інтерактивних блокнотів Jupyter. Готові програмні рішення допоможуть зрозуміти, як екосистема Python формує технологічний фундамент для фінансової індустрії.

---

*Науково-популярне видання*

**Хілпінш, Ів**

## **Python для фінансових розрахунків**

**2-е видання**

**(Рос. мовою)**

*Зав. редакцією В.Р. Гінзбург*

Із загальних питань звертайтеся до видавництва “Діалектика” за адресою:  
[info@dialektika.com](mailto:info@dialektika.com), <http://www.dialektika.com>

Підписано до друку 08.04.2021. Формат 60х90/16

Ум. друк. арк. 50,0. Обл.-вид. арк. 30,6

Зам. № 21-0857

Видавець ТОВ “Комп’ютерне видавництво “Діалектика”

03164, м. Київ, вул. Генерала Наумова, буд. 23-Б.

Свідоцтво суб’єкта видавничої справи ДК № 6758 від 16.05.2019.

Надруковано ТОВ “АЛЬФА ГРАФІК ”

03067, м. Київ, вул. Машинобудівна, 42

Свідоцтво суб’єкта видавничої справи ДК № 6838 від 09.07.2019.

# PYTHON И МАШИННОЕ ОБУЧЕНИЕ МАШИННОЕ И ГЛУБОКОЕ ОБУЧЕНИЕ С ИСПОЛЬЗОВАНИЕМ PYTHON, SCIKIT-LEARN И TENSORFLOW 2 3-Е ИЗДАНИЕ

**Себастьян Рашка  
Вахид Мирджалили**



[www.dialektika.com](http://www.dialektika.com)

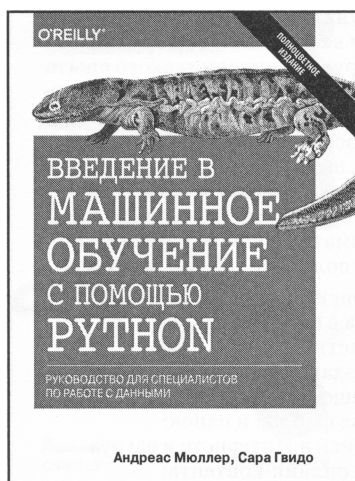
**ISBN 978-5-907203-57-0**

Прикладное машинное обучение с прочным теоретическим фундаментом. Новое издание пересмотрено и расширено с целью охвата TensorFlow 2, порождающих состязательных сетей (GAN) и обучения с подкреплением. Книга является всеобъемлющим руководством по машинному и глубокому обучению с использованием языка Python. Она служит как пошаговым учебным пособием, так и справочником, к которому вы постоянно будете возвращаться в ходе построения систем машинного обучения. Книга наполнена четкими пояснениями, визуальными представлениями и работающими примерами, детально раскрывая все важные методики машинного обучения. В то время как некоторые книги учат вас следовать инструкциям, Рашка и Мирджалили излагают принципы, лежащие в основе машинного обучения, что позволит вам самостоятельно строить модели и приложения. Обновленное с учетом библиотеки TensorFlow 2.0 третье издание предлагает читателям ознакомиться с ее новыми средствами API-интерфейса Keras, а также с последними добавлениями в scikit-learn.

**в продаже**

# ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

**Андреас Мюллер  
Сара Гвидо**



[www.williamspublishing.com](http://www.williamspublishing.com)

Эта полноцветная книга — отличный источник информации для каждого, кто собирается использовать машинное обучение на практике. Ныне машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, и не следует думать, что эта область — прерогатива исключительно крупных компаний с мощными командами аналитиков. Эта книга научит вас практическим способам построения систем МО, даже если вы еще новичок в этой области. В ней подробно объясняются все этапы, необходимые для создания успешного проекта машинного обучения, с использованием языка Python и библиотек scikit-learn, NumPy и matplotlib. Авторы сосредоточили свое внимание исключительно на практических аспектах применения алгоритмов машинного обучения, оставив за рамками книги их математическое обоснование. Данная книга адресована специалистам, решающим реальные задачи, а поскольку область применения методов МО практически безгранична, прочитав эту книгу, вы сможете собственными силами построить действующую систему машинного обучения в любой научной или коммерческой сфере.

**ISBN 978-5-9908910-8-1**

**в продаже**



# АВТОМАТИЗАЦИЯ РУТИННЫХ ЗАДАЧ С ПОМОЩЬЮ PYTHON

практическое руководство  
для начинающих

*Эл Свейгарт*



[www.williamspublishing.com](http://www.williamspublishing.com)

Книга научит вас использовать Python для написания программ, способных в считанные секунды сделать то, на что раньше у вас уходили часы ручного труда, причем никакого опыта программирования от вас не требуется. Как только вы овладеете основами программирования, вы сможете создавать программы на языке Python, которые будут без труда выполнять в автоматическом режиме различные полезные задачи, такие как:

- поиск определенного текста в файле или в множестве файлов;
- создание, обновление, перемещение и переименование файлов и папок;
- поиск в Интернете и загрузка онлайн-контента;
- обновление и форматирование данных в электронных таблицах Excel любого размера;
- разбиение, слияние, разметка водяными знаками и шифрование PDF-документов;
- рассылка напоминаний в виде сообщений электронной почты или текстовых уведомлений;
- заполнение онлайн-форм.

ISBN 978-5-6040724-2-4

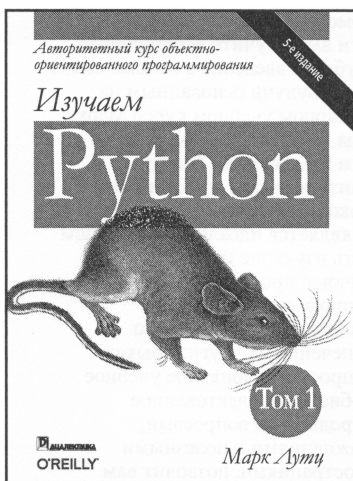
в продаже

# ИЗУЧАЕМ PYTHON

## ТОМ 1

### 5-Е ИЗДАНИЕ

**Марк Лутц**



[www.williamspublishing.com](http://www.williamspublishing.com)

С помощью этой практической книги вы получите всестороннее и глубокое введение в основы языка Python. Будучи основанным на популярном учебном курсе Марка Лутца, обновленное 5-е издание книги поможет вам быстро научиться писать эффективный высококачественный код на Python. Она является идеальным способом начать изучение Python, будь вы новичок в программировании или профессиональный разработчик программного обеспечения на других языках. Это простое и понятное учебное пособие, укомплектованное контрольными вопросами, упражнениями и полезными иллюстрациями, позволит вам освоить основы линеек Python 3.X и 2.X. Вы также ознакомитесь с расширенными возможностями языка, получившими широкое распространение в коде Python.

**ISBN 978-5-907144-52-1**

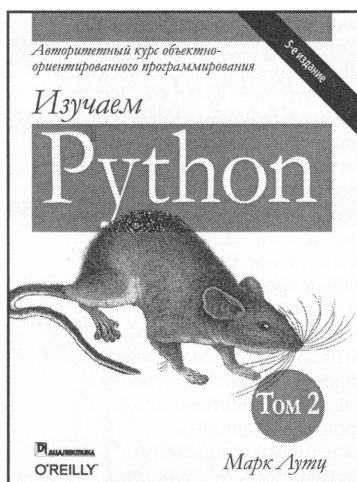
**в продаже**

# ИЗУЧАЕМ PYTHON

## ТОМ 2

### 5-Е ИЗДАНИЕ

**Марк Лутц**



[www.williamspublishing.com](http://www.williamspublishing.com)

С помощью этой практической книги вы получите всестороннее и глубокое введение в основы языка Python. Будучи основанным на популярном учебном курсе Марка Лутца, обновленное 5-е издание книги поможет вам быстро научиться писать эффективный высококачественный код на Python. Она является идеальным способом начать изучение Python, будь вы новичок в программировании или профессиональный разработчик программного обеспечения на других языках. Это простое и понятное учебное пособие, укомплектованное контрольными вопросами, упражнениями и полезными иллюстрациями, позволит вам освоить основы линеек Python 3.X и 2.X. Вы также ознакомитесь с расширенными возможностями языка, получившими широкое распространение в коде Python.

**ISBN 978-5-907144-53-8**

**в продаже**

# Python для финансовых расчетов

Python стал языком выбора для разработки финансовых приложений, управляемых данными, и систем искусственного интеллекта. Крупные инвестиционные банки и хедж-фонды все активнее реализуют свои базовые платформы трейдинга и управления рисками с использованием экосистемы Python. В новом издании книги разработчики и финансовые аналитики узнают, как применять различные инструменты Python для создания финансовых приложений и систем алгоритмической торговли.

Все примеры книги написаны на Python 3 и доступны в виде интерактивных блокнотов Jupyter. Готовые программные решения помогут понять, как экосистема Python формирует технологический фундамент для финансовой индустрии.

**“Понятный синтаксис, простота интеграции с C/C++ и наличие большого количества математических пакетов делают Python основным инструментом финансовых аналитиков. Он стремительно становится стандартом де-факто в данной области, вытесняя большинство других языков программирования.”**

Кират Сингх,  
Beacon Platform, Inc.

## ОСНОВНЫЕ ТЕМЫ КНИГИ

- **Python и финансовые вычисления.** Применение Python для интерактивного финансового анализа и разработки финансовых приложений
- **Основы Python.** Типы данных и структуры Python, библиотеки NumPy и pandas, объектно-ориентированное программирование
- **Обработка и анализ финансовых данных.** Обработка финансовых временных рядов, операции ввода-вывода, стохастические методы и алгоритмы машинного обучения
- **Алгоритмическая торговля.** Применение Python для внедрения автоматизированных систем алгоритмической торговли
- **Анализ деривативов.** Разработка гибкого и производительного программного пакета, предназначенного для оценки опционов, включая управление рисками

**Ив Хилпиш** — основатель проекта *The Python Quants*, ориентированного на применение программных продуктов с открытым исходным кодом в финансовом анализе, искусственном интеллекте и алгоритмической торговле. Он также возглавляет компанию *The AI Machine*, которая занимается разработкой стандартизированных систем алгоритмической торговли на базе искусственного интеллекта, и руководит первой обучающей онлайн-программой, нацеленной на получение университетского сертификата по алгоритмической торговле на Python.

Все иллюстрации к книге в цветном варианте доступны по адресу:  
<http://go.dialektika.com/pythonfinance>

Категория: компьютерные технологии/Python/финансы

ISBN 978-617-7874-29-3



Комп'ютерне видавництво  
"ДІАЛЕКТИКА"  
[www.dialektika.com](http://www.dialektika.com)

[www.oreilly.com](http://www.oreilly.com)