

ДЭВИД
Бизли

Python

ИСЧЕРПЫВАЮЩЕЕ
РУКОВОДСТВО



RESCUER

ДЛЯ ПРАКТИКУЮЩИХ
ПРОГРАММИСТОВ



Python Distilled

David M. Beazley

◆◆Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Python

ИСЧЕРПЫВАЮЩЕЕ РУКОВОДСТВО

Дэвид Бизли



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.43
Б59

Бизли Дэвид

Б59 Python. Исчерпывающее руководство. — СПб.: Питер, 2023. — 368 с. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1956-1

Разнообразие возможностей современного Python становится испытанием для разработчиков всех уровней. Как программисту на старте карьеры понять, с чего начать, чтобы это испытание не стало для него непосильным? Как опытному разработчику Python понять, эффективен или нет его стиль программирования? Как перейти от изучения отдельных возможностей к мышлению на Python на более глубоком уровне? «Python. Исчерпывающее руководство» отвечает на эти, а также на многие другие актуальные вопросы.

Эта книга делает акцент на основополагающих возможностях Python (3.6 и выше), а примеры кода демонстрируют «механику» языка и учат структурировать программы, чтобы их было проще читать, тестировать и отлаживать. Дэвид Бизли знакомит нас со своим уникальным взглядом на то, как на самом деле работает этот язык программирования.

Перед вами практическое руководство, в котором компактно изложены такие фундаментальные темы программирования, как абстракции данных, управление программной логикой, структура программ, функции, объекты и модули, лежащие в основе проектов Python любого масштаба.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0134173276 англ.
ISBN 978-5-4461-1956-1

© 2022 Pearson Education, Inc.
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Для профессионалов», 2022

Оглавление

| | |
|--|-----------|
| Предисловие | 13 |
| Благодарности..... | 15 |
| Об авторе | 15 |
| От издательства..... | 15 |
| Глава 1. Основы Python | 16 |
| 1.1. Запуск Python | 16 |
| 1.2. Программы Python | 17 |
| 1.3. Прimitives, переменные и выражения..... | 18 |
| 1.4. Арифметические операторы..... | 20 |
| 1.5. Условные команды и управление программной логикой | 23 |
| 1.6. Строки..... | 25 |
| 1.7. Файловый ввод и вывод | 28 |
| 1.8. Списки | 30 |
| 1.9. Кортежи..... | 32 |
| 1.10. Множества..... | 34 |
| 1.11. Словари | 35 |
| 1.12. Перебор и циклы..... | 39 |
| 1.13. Функции | 40 |
| 1.14. Исключения..... | 42 |
| 1.15. Завершение программы..... | 44 |
| 1.16. Объекты и классы..... | 44 |
| 1.17. Модули | 48 |
| 1.18. Написание сценариев | 51 |
| 1.19. Пакеты | 52 |

| | |
|---|----|
| 1.20. Структура приложения | 53 |
| 1.21. Управление сторонними пакетами | 54 |
| 1.22. Python подстраивается под ваши запросы..... | 56 |

Глава 2. Операторы, выражения и обработка данных.....57

| | |
|--|----|
| 2.1. Литералы..... | 57 |
| 2.2. Выражения и адреса памяти..... | 58 |
| 2.3. Стандартные операторы | 59 |
| 2.4. Присваивание на месте | 61 |
| 2.5. Сравнение объектов | 62 |
| 2.6. Операторы порядкового сравнения..... | 63 |
| 2.7. Логические выражения и квазиистинность..... | 64 |
| 2.8. Условные выражения..... | 65 |
| 2.9. Операции с итерируемыми объектами..... | 66 |
| 2.10. Операции с последовательностями..... | 68 |
| 2.11. Операции с изменяемыми последовательностями..... | 71 |
| 2.12. Операции с множествами | 72 |
| 2.13. Операции с отображениями..... | 73 |
| 2.14. Включения списков, множеств и словарей..... | 74 |
| 2.15. Выражения-генераторы..... | 77 |
| 2.16. Оператор атрибута (.)..... | 78 |
| 2.17. Оператор вызова функции ()..... | 79 |
| 2.18. Порядок вычисления..... | 79 |
| 2.19. Напоследок: тайная жизнь данных..... | 81 |

Глава 3. Структура программы и управление последовательностью выполнения82

| | |
|--|----|
| 3.1. Структура программы и выполнение | 82 |
| 3.2. Условное выполнение | 83 |
| 3.3. Циклы и перебор | 83 |
| 3.4. Исключения | 87 |
| 3.4.1. Иерархия исключений..... | 91 |
| 3.4.2. Исключения и последовательность выполнения..... | 93 |
| 3.4.3. Определение новых исключений..... | 94 |
| 3.4.4. Цепочки исключений..... | 95 |

| | |
|--|------------|
| 3.4.5. Трассировка исключений..... | 98 |
| 3.4.6. Рекомендации по обработке ошибок..... | 98 |
| 3.5. Менеджеры контекста и команда with | 100 |
| 3.6. Команды assert и __debug__ | 102 |
| 3.7. Напоследок..... | 103 |
| Глава 4. Объекты, типы и протоколы | 105 |
| 4.1. Важнейшие концепции | 105 |
| 4.2. Идентификатор объекта и его тип | 106 |
| 4.3. Подсчет ссылок и сбор мусора | 108 |
| 4.4. Ссылки и копии | 109 |
| 4.5. Представление и вывод объектов..... | 111 |
| 4.6. Первоклассные объекты | 112 |
| 4.7. Использование None для необязательных или отсутствующих данных | 114 |
| 4.8. Протоколы объектов и абстракции данных..... | 114 |
| 4.9. Протокол объектов | 116 |
| 4.10. Числовой протокол | 117 |
| 4.11. Протокол сравнения | 120 |
| 4.12. Протоколы преобразования..... | 122 |
| 4.13. Протокол контейнера | 124 |
| 4.14. Протокол итераций | 125 |
| 4.15. Протокол атрибутов..... | 127 |
| 4.16. Протокол функций..... | 127 |
| 4.17. Протокол менеджера контекста..... | 128 |
| 4.18. Напоследок: о коде Python | 129 |
| Глава 5. Функции | 130 |
| 5.1. Определения функций | 130 |
| 5.2. Аргументы по умолчанию..... | 130 |
| 5.3. Функции с переменным количеством аргументов..... | 131 |
| 5.4. Ключевые аргументы | 132 |
| 5.5. Функции с переменным числом ключевых аргументов..... | 133 |
| 5.6. Функции, принимающие любой ввод..... | 134 |
| 5.7. Только позиционные аргументы..... | 134 |

| | |
|--|-----|
| 5.8. Имена, строки документации и аннотации типов | 135 |
| 5.9. Применение функций и передача параметров..... | 137 |
| 5.10. Возвращаемые значения..... | 139 |
| 5.11. Обработка ошибок..... | 140 |
| 5.12. Правила масштабирования | 141 |
| 5.13. Рекурсия | 144 |
| 5.14. Лямбда-функции..... | 144 |
| 5.15. Функции высшего порядка | 146 |
| 5.16. Передача аргументов функциям обратного вызова..... | 149 |
| 5.17. Возвращение результатов из обратных вызовов..... | 153 |
| 5.18. Декораторы | 155 |
| 5.19. Отображение, фильтрация и свертка..... | 159 |
| 5.20. Интроспекция, атрибуты и сигнатуры..... | 160 |
| 5.21. Анализ среды..... | 163 |
| 5.22. Динамическое выполнение и создание кода | 165 |
| 5.23. Асинхронные функции и await | 167 |
| 5.24. Напоследок: о функциях и композиции | 169 |

Глава 6. Генераторы 171

| | |
|--|-----|
| 6.1. Генераторы и yield | 171 |
| 6.2. Перезапускаемые генераторы..... | 174 |
| 6.3. Делегирование..... | 175 |
| 6.4. Практическое использование генераторов | 176 |
| 6.5. Расширенные генераторы и выражения yield | 179 |
| 6.6. Применение расширенных генераторов | 180 |
| 6.7. Генераторы и их связь с await..... | 184 |
| 6.8. Напоследок: краткая история и возможности генераторов | 185 |

Глава 7. Классы и объектно-ориентированное программирование 186

| | |
|-----------------------------------|-----|
| 7.1. Объекты | 186 |
| 7.2. Команда class..... | 187 |
| 7.3. Экземпляры..... | 189 |
| 7.4. Обращение к атрибутам..... | 190 |
| 7.5. Правила масштабирования..... | 191 |

| | |
|---|------------|
| 7.6. Перегрузка операторов и протоколы | 192 |
| 7.7. Наследование..... | 193 |
| 7.8. Отказ от наследования в пользу композиции..... | 197 |
| 7.9. Замена наследования функциями..... | 200 |
| 7.10. Динамическая и утиная типизации..... | 201 |
| 7.11. Опасность наследования от встроенных типов..... | 201 |
| 7.12. Переменные и методы класса | 203 |
| 7.13. Статические методы | 207 |
| 7.14. О паттернах проектирования | 210 |
| 7.15. Инкапсуляция данных и приватные атрибуты | 211 |
| 7.16. Аннотации типов..... | 213 |
| 7.17. Свойства..... | 214 |
| 7.18. Типы, интерфейсы и абстрактные базовые классы..... | 218 |
| 7.19. Множественное наследование, интерфейсы и примеси | 222 |
| 7.20. Диспетчеризация вызовов в зависимости от типа | 228 |
| 7.21. Декораторы классов..... | 230 |
| 7.22. Контролируемое наследование..... | 233 |
| 7.23. Жизненный цикл объектов и управление памятью | 235 |
| 7.24. Слабые ссылки..... | 240 |
| 7.25. Внутреннее представление объектов и связывание атрибутов | 242 |
| 7.26. Прокси, обертки и делегирование..... | 244 |
| 7.27. Сокращение затрат памяти и __slots__ | 247 |
| 7.28. Дескрипторы..... | 248 |
| 7.29. Процесс определения класса | 251 |
| 7.30. Динамическое создание класса..... | 253 |
| 7.31. Метаклассы | 254 |
| 7.32. Встроенные объекты для экземпляров и классов | 259 |
| 7.33. Напоследок: будьте проще..... | 260 |
| Глава 8. Модули и пакеты | 262 |
| 8.1. Модули и команда import..... | 262 |
| 8.2. Кеширование модулей..... | 265 |
| 8.3. Импортирование отдельных имен из модуля | 265 |
| 8.4. Циклический импорт..... | 268 |
| 8.5. Перезагрузка и выгрузка модулей | 269 |

| | |
|--|------------|
| 8.6. Компиляция модулей | 271 |
| 8.7. Путь поиска модулей | 272 |
| 8.8. Выполнение в качестве основной программы | 272 |
| 8.9. Пакеты..... | 273 |
| 8.10. Импорт из пакета..... | 275 |
| 8.11. Выполнение подмодуля пакета в качестве сценария..... | 276 |
| 8.12. Управление пространством имен пакета | 277 |
| 8.13. Управление экспортом пакетов..... | 278 |
| 8.14. Данные пакетов | 280 |
| 8.15. Объекты модулей..... | 281 |
| 8.16. Развертывание пакетов Python..... | 282 |
| 8.17. Начинайте с пакета..... | 284 |
| 8.18. Напоследок: будьте проще | 285 |
| Глава 9. Ввод/вывод..... | 286 |
| 9.1. Представление данных..... | 286 |
| 9.2. Кодирование и декодирование текста | 287 |
| 9.3. Форматирование текста и байтов..... | 289 |
| 9.4. Чтение параметров командной строки | 294 |
| 9.5. Переменные среды..... | 296 |
| 9.6. Файлы и объекты файлов | 296 |
| 9.6.1. Имена файлов..... | 297 |
| 9.6.2. Режимы открытия файлов | 298 |
| 9.6.3. Буферизация ввода/вывода..... | 299 |
| 9.6.4. Кодировка текстового режима..... | 300 |
| 9.6.5. Обработка строк текста в текстовом режиме | 300 |
| 9.7. Уровни абстракции ввода/вывода | 301 |
| 9.7.1. Методы файлов | 302 |
| 9.8. Стандартный ввод, вывод и поток ошибок | 305 |
| 9.9. Каталоги | 306 |
| 9.10. Функция print() | 307 |
| 9.11. Генерация вывода..... | 307 |
| 9.12. Потребление входных данных..... | 308 |
| 9.13. Сериализация объектов..... | 310 |
| 9.14. Блокирующие операции и параллелизм..... | 311 |

| | |
|---|-----|
| 9.14.1. Неблокирующий ввод/вывод | 312 |
| 9.14.2. Опрос каналов ввода/вывода | 313 |
| 9.14.3. Потоки..... | 314 |
| 9.14.4. Параллельное выполнение в asyncio | 315 |
| 9.15. Модули стандартной библиотеки | 315 |
| 9.15.1. Модуль asyncio..... | 316 |
| 9.15.2. Модуль binascii | 317 |
| 9.15.3. Модуль cgi | 317 |
| 9.15.4. Модуль configparser..... | 318 |
| 9.15.5. Модуль csv..... | 319 |
| 9.15.6. Модуль errno..... | 320 |
| 9.15.7. Модуль fcntl | 321 |
| 9.15.8. Модуль hashlib | 321 |
| 9.15.9. Пакет http..... | 322 |
| 9.15.10. Модуль io | 322 |
| 9.15.11. Модуль json..... | 323 |
| 9.15.12. Модуль logging..... | 324 |
| 9.15.13. Модуль os..... | 324 |
| 9.15.14. Модуль os.path | 325 |
| 9.15.15. Модуль pathlib | 326 |
| 9.15.16. Модуль re..... | 327 |
| 9.15.17. Модуль shutil..... | 327 |
| 9.15.18. Модуль select..... | 328 |
| 9.15.19. Модуль smtplib | 329 |
| 9.15.20. Модуль socket..... | 329 |
| 9.15.21. Модуль struct | 331 |
| 9.15.22. Модуль subprocess | 332 |
| 9.15.23. Модуль tempfile | 333 |
| 9.15.24. Модуль textwrap..... | 333 |
| 9.15.25. Модуль threading | 334 |
| 9.15.26. Модуль time | 336 |
| 9.15.27. Пакет urllib | 337 |
| 9.15.28. Модуль unicodedata | 338 |
| 9.15.29. Пакет xml | 339 |
| 9.16. Напоследок | 340 |

Глава 10. Встроенные функции и стандартная библиотека..... 341

| | |
|---|-----|
| 10.1. Встроенные функции | 341 |
| 10.2. Встроенные исключения | 360 |
| 10.2.1. Базовые классы исключений | 360 |
| 10.2.2. Атрибуты исключений | 361 |
| 10.2.3. Предварительно определенные классы исключений | 361 |
| 10.3. Стандартная библиотека | 364 |
| 10.3.1. Модуль collections..... | 364 |
| 10.3.2. Модуль datetime | 364 |
| 10.3.3. Модуль itertools..... | 364 |
| 10.3.4. Модуль inspect | 364 |
| 10.3.5. Модуль math | 365 |
| 10.3.6. Модуль os..... | 365 |
| 10.3.7. Модуль random | 365 |
| 10.3.8. Модуль re | 365 |
| 10.3.9. Модуль shutil..... | 365 |
| 10.3.10. Модуль statistics..... | 365 |
| 10.3.11. Модуль sys..... | 365 |
| 10.3.12. Модуль time | 365 |
| 10.3.13. Модуль turtle..... | 366 |
| 10.3.14. Модуль unittest..... | 366 |
| 10.4. Напоследок: использование встроенных модулей..... | 366 |

Предисловие

Я написал книгу *Python Essential Reference* более 20 лет назад. Тогда Python был не таким развитым языком и к нему прилагался полезный инструментарий в виде стандартной библиотеки. Все это вполне укладывалось в голову. Книга отражала особенности той эпохи. Она была небольшой, и ее можно было взять с собой, чтобы заняться написанием кода Python на необитаемом острове или в тайном убежище. В трех последующих переизданиях книга осталась компактным, но полным справочником по языку. Если вы собираетесь программировать на Python в отпуске, почему бы не использовать все его возможности?

Прошло уже более десяти лет с момента публикации последнего издания, и мир Python сильно изменился. Python перестал быть нишевым и стал одним из самых популярных языков программирования в мире. Программистам Python доступен огромный объем информации в форме современных редакторов, IDE, блокнотов Jupyter, веб-страниц и т. д. Вряд ли кому-нибудь захочется обращаться к справочнику, когда любую информацию можно вызвать за несколько кликов.

Простота получения информации и размеры мира Python создают другую проблему. Если вы только начинаете изучать Python или собираетесь решать новую задачу, выбрать отправную точку может быть непросто. Еще может быть трудно отделить функциональность разных инструментов от базовых возможностей языка. Эти проблемы подтолкнули меня к написанию книги.

Эта книга посвящена программированию на языке Python. Я не пытался документировать все, что можно сделать или было сделано на Python, а постарался представить современное ядро языка без всего второстепенного. Основой для издания стал мой многолетний опыт преподавания Python ученым, инженерам и профессиональным программистам. Это следствие написания программных библиотек и попыток выйти за рамки возможностей Python для поиска самых полезных аспектов.

Материал касается самого программирования Python. В книге рассматриваются приемы абстракции, структура программ, данные, функции, объекты,

модули и т. д. — темы, полезные для программистов, работающих над любыми проектами Python. Чистый справочный материал, который можно легко получить в IDE (списки функций, имена команд, аргументы и т. д.), обычно опускается. Я также осознанно решил не описывать стремительно меняющийся мир инструментов Python — редакторов, IDE, средств развертывания и других сопутствующих тем.

Кому-то это покажется спорным, но я не рассматриваю средства языка, связанные с управлением крупномасштабными программными проектами. Да, Python иногда используется для больших и серьезных проектов из миллионов строк кода. Такие приложения требуют специальных инструментов, методов проектирования и функциональности. Все это не поместится в такой маленькой книге. Честнее будет ответить, что я не использовал Python для написания таких приложений — и вам не советую (по крайней мере не в качестве хобби).

При написании книги всегда есть предел для постоянно развивающихся возможностей языка. Эта книга была написана в эпоху Python 3.9. Поэтому в ней нет многих крупных дополнений, запланированных для следующих выпусков, например структурированного поиска по шаблону. Для них тоже найдется свое место и время.

Для меня очень важно, чтобы программирование оставалось интересным. Надеюсь, моя книга не только поможет вам увереннее использовать этот язык, но и вместит часть волшебства, вдохновляющего людей использовать Python для исследования космоса, отправки управляемых аппаратов на Марс и полива белок из шланга на заднем дворе.

БЛАГОДАРНОСТИ

Я хочу поблагодарить научных редакторов Шона Брауна, Софи Табак и Пита Фейна за их полезные замечания. Хочу сказать спасибо редактору Дебре Уильямс Коули, с которой я уже давно работаю, за ее участие в этом и предыдущих проектах. Многие студенты, посещавшие мои занятия, косвенно повлияли на темы, рассмотренные в книге. И конечно, я благодарю Полу, Томаса и Льюиса за их поддержку и любовь.

ОБ АВТОРЕ

Дэвид Бизли — автор книг *Python Essential Reference*, 4-е издание (Addison-Wesley, 2010) и *Python Cookbook*¹, 3-е издание (O'Reilly, 2013). Сейчас ведет учебные курсы повышения квалификации в своей компании Dabeaz LLC (www.dabeaz.com). Он пишет на Python и преподает его с 1996 года.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

¹ Бизли Д. Python. Книга рецептов. — 2020.

ГЛАВА 1

Основы Python

В этой главе рассмотрены основные особенности языка Python — переменные, типы данных, выражения, функции, классы, ввод/вывод, а также управление логикой выполнения программы. Глава завершается обсуждением модулей, написанием сценариев и пакетов, советами по организации больших программ. Здесь вы не найдете исчерпывающих описаний каждой возможности или инструментов, необходимых для более крупных проектов на Python. Опытные программисты смогут почерпнуть нужную информацию для написания полнофункциональных программ. Новичкам желательно опробовать примеры в простой среде — например, в окне терминала или текстовом редакторе.

1.1. ЗАПУСК PYTHON

Программы на языке Python выполняются интерпретатором. Есть много разных сред, в которых он может работать, — в IDE, браузерах или окнах терминала. Но интерпретатор в первую очередь — текстовое приложение, которое можно запустить командой `python` из командной строки, такой как `bash`. Python 2 может быть установлен на одном компьютере с Python 3. Поэтому вам придется ввести команду `python2` или `python3` для выбора версии. В книге предполагается, что вы используете Python 3.8 или более новую версию.

При запуске интерпретатора появляется приглашение. В нем можно вводить программы в режиме REPL (Read-Evaluation-Print Loop, то есть «цикл чтение/вычисление/печать»). Например, в следующем примере интерпретатор выводит сообщение об авторских правах и отображает приглашение `>>>`, где пользователь вводит известное приветствие `Hello World`:

```
Python 3.8.0 (default, Feb 3 2019, 05:53:21)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> print('Hello World')
Hello World
>>>
```

В некоторых средах приглашение может выглядеть иначе. Следующий вывод получен в `ipython` (альтернативная оболочка для Python):

```
Python 3.8.0 (default, Feb 4, 2019, 07:39:16)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print('Hello World')
Hello World

In [2]:
```

Независимо от конкретной формы вывода общий принцип остается один. Вы вводите команду, она выполняется, и вы немедленно получаете результат.

Интерактивный режим Python — одна из самых полезных возможностей языка. Вы можете ввести любую команду и немедленно увидеть результат. Такой режим полезен для отладки и экспериментов. Многие используют интерактивный режим Python как настольный калькулятор:

```
>>> 6000 + 4523.50 + 134.25
10657.75
>>> _ + 8192.75
18850.5
>>>
```

Когда Python используют в интерактивном режиме, переменная `_` содержит результат последней операции. Это удобно, если вы хотите задействовать ее в дальнейшем. Такая переменная определяется только в интерактивном режиме, не пытайтесь использовать ее в сохраняемых программах.

Для выхода из интерактивного интерпретатора введите команду `quit()` или символ EOF (End Of File). В системе UNIX это сочетание клавиш `Ctrl+D`, а в Windows — `Ctrl+Z`.

1.2. ПРОГРАММЫ PYTHON

Если вы хотите создать программу, которую можно запускать многократно, поместите операторы в текстовый файл. Например:

```
# hello.py
print('Hello World')
```

Исходные файлы Python — это текстовые файлы в кодировке UTF-8, обычно имеющие суффикс `.py`. Символ `#` — это комментарий, продолжающийся до конца строки. Международные символы («Юникод») могут свободно применяться в исходном коде при условии использования кодировки UTF-8 (она выбирается по умолчанию в большинстве редакторов, но, если вы не уверены, проверьте конфигурацию редактора).

Чтобы выполнить файл `hello.py`, укажите его имя в командной строке интерпретатора:

```
shell % python3 hello.py
Hello World
shell %
```

Интерпретатор часто указывается в первой строке программы символами `#!`:

```
#!/usr/bin/env python3
print('Hello World')
```

В UNIX вы сможете запустить программу командой `hello.py` в командной оболочке. Например, `chmod +x hello.py` (если файлу были предоставлены разрешения выполнения).

В Windows для запуска можно дважды щелкнуть на файле `.py` или ввести имя программы в поле команды **Выполнить** меню **Пуск**. Строка `#!`, если она есть, используется для выбора версии интерпретатора (Python 2 или 3). Программа выполняется в консольном окне, которое исчезает сразу после завершения программы — часто до того, как вы успеете прочитать ее вывод. Для отладки лучше запускать программу в среде разработки Python.

Интерпретатор выполняет команды по порядку, пока не достигнет конца входного файла. В этот момент программа прекращает работу, а интерпретатор Python завершается.

1.3. ПРИМИТИВЫ, ПЕРЕМЕННЫЕ И ВЫРАЖЕНИЯ

Python — это набор примитивных типов — целых чисел, чисел с плавающей точкой, строк и т. д.:

```
42          # int
4.2         # float
'forty-two' # str
True        # bool
```

Переменная — имя, указывающее на значение. Значение представляет объект некоторого типа:

```
x = 42
```

Иногда тип явно указывается для имени:

```
x: int = 42
```

Тип — лишь подсказка, упрощающая чтение кода. Он может использоваться сторонними инструментами проверки кода. В остальных случаях он полностью игнорируется. Указание типа никак не мешает вам присвоить переменной значение другого типа.

Выражение — это комбинация примитивов, имен и операторов, в результате вычисления которой будет получено некоторое значение:

```
2 + 3 * 4 # -> 14
```

Следующая программа использует переменные и выражения для вычисления сложных процентов:

```
# interest.py

principal = 1000      # Исходная сумма
rate = 0.05           # Процентная ставка
numyears = 5          # Количество лет
year = 1
while year <= numyears:
    principal = principal * (1 + rate)
    print(year, principal)
    year += 1
```

При выполнении программа выдает следующий результат:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.5062500000001
5 1276.2815625000003
```

Команда `while` проверяет условное выражение, следующее сразу за ключевым словом. В случае истинности проверяемого условия выполняется тело команды `while`. Затем это условие проверяется повторно и тело выполняется снова, пока условие не станет ложным. Тело цикла обозначается отступами. Так, три оператора, следующие за `while` в `interest.py`, выполняются при каждой итерации. В спецификации Python не указана величина отступов.

Важно лишь, чтобы отступ был единым в границах блока. Чаще всего используются отступы из четырех пробелов на один уровень.

Один из недостатков программы `interest.py` — не очень красивый вывод. Для его улучшения можно выровнять столбцы по правому краю и ограничить точность вывода чисел двумя знаками в дробной части. Измените функцию `print()`, чтобы в ней использовалась так называемая *f-строка*:

```
print(f'{year:>3d} {principal:0.2f}')
```

В *f-строках* могут вычисляться выражения и имена переменных. Для этого они заключаются в фигурные скобки. К каждому заменяемому элементу может быть присоединен спецификатор формата. Так, `'>3d'` обозначает трехзначное десятичное число, выравниваемое по правому краю, `'0.2f'` обозначает число с плавающей точкой, выводимое с двумя знаками точности. Больше информации о кодах форматирования вы найдете в главе 9.

Теперь вывод программы выглядит так:

```
1 1050.00
2 1102.50
3 1157.62
4 1215.51
5 1276.28
```

1.4. АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ

Python поддерживает стандартный набор математических операторов (табл. 1.1). Их значение такое же, как и в большинстве языков программирования.

Таблица 1.1. Арифметические операторы

| ОПЕРАТОР | ОПИСАНИЕ |
|---------------------|--------------------------------------|
| <code>x + y</code> | Сложение |
| <code>x - y</code> | Вычитание |
| <code>x * y</code> | Умножение |
| <code>x / y</code> | Деление |
| <code>x // y</code> | Целочисленное деление |
| <code>x ** y</code> | Возведение в степень (x в степень y) |
| <code>x % y</code> | Остаток (от деления x на y) |
| <code>-x</code> | Унарный минус |
| <code>+x</code> | Унарный плюс |

Применение оператора деления (/) к целым числам дает результат с плавающей точкой. Так, результат выражения $7/4$ равен 1.75. Оператор целочисленного деления // (его еще называют делением с остатком) усекает результат до целого числа и работает как с целыми числами, так и с числами с плавающей точкой. Оператор по модулю возвращает остаток от деления $x // y$. Например, результат выражения $7 \% 4$ равен 3. Для чисел с плавающей точкой оператор по модулю возвращает остаток от деления $x // y$ в виде числа с плавающей точкой, то есть $x - (x // y) * y$.

Встроенные функции из табл. 1.2 реализуют некоторые часто используемые числовые операции.

Таблица 1.2. Распространенные математические функции

| ФУНКЦИЯ | ОПИСАНИЕ |
|---------------------------------|---|
| <code>abs(x)</code> | Модуль (абсолютная величина) |
| <code>divmod(x,y)</code> | Возвращает $(x // y, x \% y)$ |
| <code>pow(x,y [,modulo])</code> | $(x ** y) \% modulo$ |
| <code>round(x, [n])</code> | Округляет до ближайшего кратного 10 в степени n |

Функция `round()` реализует так называемое банковское округление. Если округляемое значение находится на равных расстояниях от двух кратных, оно округляется до ближайшего четного кратного (например, 0,5 округляется до 0, а 1,5 — до 2,0).

С целыми числами можно использовать ряд дополнительных операторов для битовых операций (табл. 1.3).

Таблица 1.3. Битовые операторы

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--------------|-------------------------------|
| $x \ll y$ | Битовый сдвиг влево |
| $x \gg y$ | Битовый сдвиг вправо |
| $x \& y$ | Битовая операция И |
| $x y$ | Битовая операция ИЛИ |
| $x \wedge y$ | Битовый сдвиг ИСКЛЮЧАЮЩЕЕ ИЛИ |
| $\sim x$ | Битовое отрицание |

Эти операции обычно используют с двоичными целыми числами:

```
a = 0b11001001
mask = 0b11110000
x = (a & mask) >> 4 # x = 0b1100 (12)
```

В этом примере `0b11001001` — запись целого числа в двоичном виде. Его можно записать в десятичной форме `201` или шестнадцатеричной форме `0xc9`. Но, если вы работаете на уровне отдельных битов, двоичная запись помогает наглядно представить происходящее.

Суть битовых операций в том, что целые числа используют представление в дополнительном коде, а знаковый бит бесконечно распространяется влево. Если вы работаете с низкоуровневыми битовыми последовательностями, которые должны представлять целые числа для оборудования, будьте внимательны. Python не усекает биты и не поддерживает переполнение — вместо этого результат неограниченно растет. Вы сами должны следить, чтобы результат был нужного размера или усекался при необходимости.

Для сравнения чисел используются операторы сравнения (табл. 1.4).

Таблица 1.4. Операторы сравнения

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|------------------------|------------------|
| <code>x == y</code> | Равно |
| <code>x != y</code> | Не равно |
| <code>x < y</code> | Меньше |
| <code>x > y</code> | Больше |
| <code>x >= y</code> | Больше или равно |
| <code>x <= y</code> | Меньше или равно |

Результат сравнения — логическое (булевское) значение `True` или `False`.

Операторы `and`, `or` и `not` (не путайте с битовыми операциями из табл. 1.3) могут использоваться для построения более сложных логических выражений. Поведение этих операторов описано в табл. 1.5.

Таблица 1.5. Логические операторы

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|----------------------|---|
| <code>x or y</code> | Если значение <code>x</code> ложно, возвращается <code>y</code> , в противном случае возвращается <code>x</code> |
| <code>x and y</code> | Если значение <code>x</code> ложно, возвращается <code>x</code> , в противном случае возвращается <code>y</code> |
| <code>not x</code> | Если значение <code>x</code> ложно, возвращается <code>True</code> , в противном случае возвращается <code>False</code> |

Значение интерпретируется как ложное, если это `False`, `None`, числовой нуль или пустое значение. Во всех остальных случаях оно интерпретируется как истинное.

Очень часто встречаются выражения, обновляющие значение:

```
x = x + 1
y = y * n
```

В таких случаях можно использовать следующие сокращенные операции:

```
x += 1
y *= n
```

Сокращенная форма обновления может использоваться с любым из операторов `+`, `-`, `*`, `**`, `/`, `//`, `%`, `&`, `|`, `^`, `<<`, `>>`. В Python нет операторов инкремента (`++`) или декремента (`--`), встречающихся в других языках программирования.

1.5. УСЛОВНЫЕ КОМАНДЫ И УПРАВЛЕНИЕ ПРОГРАММНОЙ ЛОГИКОЙ

Команды `while`, `if` и `else` используются для повторения и условного выполнения кода:

```
if a < b:
    print('Computer says Yes')
else:
    print('Computer says No')
```

Тела операторов `if` и `else` обозначаются отступами. Наличие оператора `else` необязательно. Для создания пустой секции используйте команду `pass`:

```
if a < b:
    pass # Ничего не делать
else:
    print('Computer says No')
```

Для реализации выбора со многими вариантами предназначена команда `elif`:

```
if suffix == '.htm':
    content = 'text/html'
elif suffix == '.jpg':
    content = 'image/jpeg'
```

```

elif suffix == '.png':
    content = 'image/png'
else:
    raise RuntimeError(f'Unknown content type {suffix!r}')

```

Если значение присваивается по результатам проверки условия, используйте условное выражение:

```
maxval = a if a > b else b
```

Это то же, но короче:

```

if a > b:
    maxval = a
else:
    maxval = b

```

Иногда вы можете увидеть назначение переменной и условного оператора, объединенных с помощью оператора `:=`. Это называется выражением присваивания (или в просторечии моржом, потому что `:=` напоминает голову моржа, повернутую на 90°). Например:

```

x = 0
while (x := x + 1) < 10: # Выводит 1, 2, 3, ..., 9
    print(x)

```

Круглые скобки, в которые заключено выражение присваивания, обязательны.

Команда `break` может использоваться для преждевременного прерывания цикла. Она работает только в цикле с наибольшим уровнем вложенности. Пример:

```

x = 0
while x < 10:
    if x == 5:
        break # Прерывает цикл, переходит к выводу Done
    print(x)
    x += 1

print('Done')

```

Команда `continue` пропускает остаток тела цикла и возвращает управление к началу цикла. Пример:

```

x = 0
while x < 10:
    x += 1

```

```

    if x == 5:
        continue # Пропустить print(x), вернуться к началу цикла
    print(x)

print('Done')

```

1.6. СТРОКИ

Для определения строкового литерала заключите его в одинарные, двойные или тройные кавычки:

```

a = 'Hello World'
b = "Python is groovy"
c = '''Computer says no.'''
d = """Computer still says no."""

```

Для завершения и открытия строки должны использоваться одинаковые кавычки. Строки в тройных кавычках захватывают весь текст до завершающей тройной кавычки, в отличие от строк в одинарных и двойных кавычках, которые должны быть указаны в одной логической строке. Строки в тройных кавычках полезны, когда содержимое строкового литерала занимает несколько строк текста:

```

print('''Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.'''
)

```

Строковые литералы, следующие друг за другом, объединяются в одну строку. Так, предыдущий пример можно было записать в следующем виде:

```

print(
    'Content-type: text/html\n'
    '\n'
    '<h1> Hello World </h1>\n'
    'Click <a href="http://www.python.org">here</a>\n'
)

```

Если перед открывающей кавычкой строки находится префикс `f`, то в строке выполняется вычисление и подстановка экранированных выражений. Например, в прошлом примере для вывода результатов вычислений использовалась следующая команда:

```

print(f'{year:>3d} {principal:0.2f}')

```

И хотя здесь в строку включаются простые имена переменных, в ней могут находиться любые допустимые выражения:

```
base_year = 2020
...
print(f'{base_year + year:>4d} {principal:0.2f}')
```

Как альтернатива f-строкам для форматирования строк иногда используются метод `format()` и оператор `%`:

```
print('{0:>3d} {1:0.2f}'.format(year, principal))
print('%3d %0.2f' % (year, principal))
```

Подробнее форматирование строк рассматривается в главе 9.

Строки хранятся в виде последовательностей символов «Юникода», которые индексируются целыми числами, начиная с 0. Отрицательные индексы отсчитываются от конца строки. Длина строки `s` вычисляется функцией `len(s)`. Чтобы извлечь из строки один символ, используйте оператор индексирования `s[i]`, где `i` — индекс.

```
a = 'Hello World'
print(len(a))           # 11
b = a[4]                 # b = 'o'
c = a[-1]                # c = 'd'
```

Для извлечения подстроки используется оператор сегментации `s[i:j]`. Он извлекает из `s` все символы, индекс `k` которых лежит в диапазоне $i \leq k < j$. Если один из индексов опущен, по умолчанию используется начало или конец строки соответственно:

```
c = a[:5]                # c = 'Hello'
d = a[6:]                # d = 'World'
e = a[3:8]               # e = 'lo Wo'
f = a[-5:]               # f = 'World'
```

Строки поддерживают разные методы для выполнения операций с их содержимым. Например, `replace()` выполняет простую замену текста:

```
g = a.replace('Hello', 'Hello Cruel') # f = 'Hello Cruel World'
```

В табл. 1.6 перечислены некоторые распространенные методы строк. Здесь и далее аргументы в квадратных скобках необязательны.

Конкатенация строк выполняется оператором `+`:

```
g = a + 'ly'             # g = 'Hello Worldly'
```

Python никогда не интерпретирует содержимое строки как числовые данные. Поэтому + всегда выполняет конкатенацию строк:

```
x = '37'
y = '42'
z = x + y      # z = '3742' (конкатенация строк)
```

Таблица 1.6. Распространенные методы строк

| МЕТОД | ОПИСАНИЕ |
|---|--|
| <code>s.endswith(prefix [,start [,end]])</code> | Проверяет, завершается ли строка подстрокой <code>prefix</code> |
| <code>s.find(sub [, start [,end]])</code> | Находит первое вхождение заданной подстроки <code>sub</code> или <code>-1</code> , если строка не найдена |
| <code>s.lower()</code> | Преобразует строку к нижнему регистру |
| <code>s.replace(old, new [,maxreplace])</code> | Заменяет подстроку |
| <code>s.split([sep [,maxsplit]])</code> | Разбивает строку по разделителю <code>sep</code> . <code>maxsplit</code> — максимальное количество выполняемых разбиений |
| <code>s.startswith(prefix [,start [,end]])</code> | Проверяет, начинается ли строка с префикса <code>prefix</code> |
| <code>s.strip([chars])</code> | Удаляет начальные и конечные пропуски/символы, переданные в <code>chars</code> |
| <code>s.upper()</code> | Преобразует строку в верхний регистр |

Для математических вычислений строку `first` нужно сначала преобразовать в числовое значение функцией `int()` или `float()`:

```
z = int(x) + int(y)      # z = 79 (целочисленное сложение)
```

Для преобразования нестроковых значений в строковое представление можно воспользоваться функциями `str()`, `repr()` или `format()`:

```
s = 'The value of x is ' + str(x)
s = 'The value of x is ' + repr(x)
s = 'The value of x is ' + format(x, '4d')
```

И хотя обе функции, `str()` и `repr()`, создают строки, их вывод часто различается. `str()` выдает результат, получаемый при использовании функции `print()`, а `repr()` создает строку, которая вводится в программе для точного представления значения объекта. Например:

```
>>> s = 'hello\nworld'
>>> print(str(s))
hello
world
>>> print(repr(s))
'hello\nworld'
>>>
```

В процессе отладки для вывода обычно используется функция `repr(s)`. Она выводит больше информации о значении и его типе.

Функция `format()` преобразует одно значение в строку с применением определенного форматирования:

```
>>> x = 12.34567
>>> format(x, '0.2f')
'12.35'
>>>
```

Функции `format()` передаются те же коды форматирования, что используются с f-строками для получения отформатированного вывода. Например, предыдущий код можно заменить таким:

```
>>> f'{x:0.2f}'
'12.35'
>>>
```

1.7. ФАЙЛОВЫЙ ВВОД И ВЫВОД

Следующая программа открывает файл и построчно читает его содержимое:

```
with open('data.txt') as file:
    for line in file:
        print(line, end='') # end='' опускает лишний символ новой строки
```

Функция `open()` возвращает новый объект файла. Команда `with`, предшествующая открытию файла, объявляет блок команд (или контекст), где будет использоваться файл (`file`). При выходе управления за его пределы файл автоматически закрывается. Без команды `with` код должен выглядеть примерно так:

```
file = open('data.txt')
for line in file:
    print(line, end='') # end='' опускает лишний символ новой строки
file.close()
```

О вызове `close()` легко забыть. Лучше использовать команду `with`, которая закрывает файл за вас.

Цикл `for` построчно перебирает данные файла, пока они не закончатся. Чтобы прочитать весь файл в виде строки, используйте метод `read()`:

```
with open('data.txt') as file:
    data = file.read()
```

Если вы хотите читать большой файл по блокам, подскажите методу `read()` размер:

```
with open('data.txt') as file:
    while (chunk := file.read(10000)):
        print(chunk, end='')
```

Оператор `:=` здесь присваивает значение переменной и возвращает его, чтобы оно могло проверяться циклом `while` для прерывания. По достижении конца файла `read()` возвращает пустую строку. Другой вариант написания этой функции основан на использовании `break`:

```
with open('data.txt') as file:
    while True:
        chunk = file.read(10000)
        if not chunk:
            break
        print(chunk, end='')
```

Чтобы направить вывод программы в файл, передайте этот файл в аргументе функции `print()`:

```
with open('out.txt', 'wt') as out:
    while year <= numyears:
        principal = principal * (1 + rate)
        print(f'{year:>3d} {principal:0.2f}', file=out)
        year += 1
```

Объекты файлов поддерживают и метод `write()`. Он может использоваться для записи строковых данных. Например, вызов `print()` из предыдущего примера можно было бы записать так:

```
out.write(f'{year:3d} {principal:0.2f}\n')
```

По умолчанию файлы с текстом используют кодировку UTF-8. При работе с другой кодировкой используйте дополнительный аргумент кодировки при открытии файла:

```
with open('data.txt', encoding='latin-1') as file:
    data = file.read()
```


Оператор `+` используется для конкатенации списков:

```
a = ['x', 'y'] + ['z', 'z', 'y'] # Result is ['x', 'y', 'z', 'z', 'y']
```

Пустой список можно создать двумя способами:

```
names = []      # Пустой список
names = list()  # Пустой список
```

Использование `[]` для создания пустых списков считается более приемлемым. `list` — имя класса, связанное с типом списка. На практике этот способ чаще используется при преобразованиях данных в список:

```
letters = list('Dave') # letters = ['D', 'a', 'v', 'e']
```

Обычно все элементы списка относятся к одному типу (например, список чисел или строк). Но списки могут содержать произвольные комбинации объектов Python, в том числе и других списков:

```
a = [1, 'Dave', 3.14, ['Mark', 7, 9, [100, 101]], 10]
```

Для обращения к элементам вложенных списков нужно добавить несколько операций индексирования:

```
a[1]          # Возвращает 'Dave'
a[3][2]       # Возвращает 9
a[3][3][1]    # Возвращает 101
```

Программа `pcost.py` демонстрирует чтение данных в список и выполнение простого вычисления. В нашем примере предполагается, что в списках есть значения, разделенные запятыми. Программа вычисляет сумму произведений двух столбцов:

```
# pcost.py
#
# Читает входные строки в формате 'НАЗВАНИЕ,КОЛИЧЕСТВО,ЦЕНА'
# Пример:
#
# SYM,123,456.78

import sys
if len(sys.argv) != 2:
    raise SystemExit(f'Usage: {sys.argv[0]} filename')

rows = []
with open(sys.argv[1], 'rt') as file:
    for line in file:
        rows.append(line.split(','))
```

```
# rows - список в форме
# [
#   ['SYM', '123', '456.78']
#   ...
# ]

total = sum([ int(row[1]) * float(row[2]) for row in rows ])
print(f'Total cost: {total:0.2f}')
```

В первой строке программы команда `import` применяется для загрузки модуля `sys` из библиотеки Python. Этот модуль используется для получения аргументов командной строки из списка `sys.argv`. Сначала программа проверяет, задано ли имя файла. Если нет, выдается исключение `SystemExit` с содержательным сообщением об ошибке. В этом сообщении `sys.argv[0]` вставляет имя выполняемой программы.

Функция `open()` использует имя файла, заданное в командной строке. Цикл `for line in file` читает каждую строку файла, преобразующуюся в небольшой список. При этом запятая используется как разделитель. Список присоединяется к `rows`. Окончательный результат `rows` — список списков (список может содержать все, что угодно, в том числе и другие списки).

Выражение `[int(row[1]) * float(row[2]) for row in rows]` строит новый список, перебирая все списки в `rows` и вычисляя произведение второго и третьего элементов. Это называется списковым включением (list comprehension). Те же вычисления можно было бы выразить более подробно:

```
values = []
for row in rows:
    values.append(int(row[1]) * float(row[2]))
total = sum(values)
```

Обычно списковые включения наиболее предпочтительны для выполнения простых вычислений. Встроенная функция `sum()` вычисляет сумму всех элементов последовательности.

1.9. КОРТЕЖИ

Чтобы создать простые структуры данных, вы можете упаковать набор значений в неизменяемый объект — кортеж (tuple). Для создания кортежа заключите группу значений в круглые скобки:

```
holding = ('GOOG', 100, 490.10)
address = ('www.python.org', 80)
```

Для полноты можно определять 0- и 1-элементные кортежи, для которых используется специальный синтаксис:

```
a = ()      # 0-элементный кортеж (пустой кортеж)
b = (item,) # 1-элементный кортеж (обратите внимание
           # на завершающую запятую)
```

Извлечь значения из кортежа можно с помощью числового индекса (как и в случае со списками). Но чаще всего кортежи распаковываются в набор переменных:

```
name, shares, price = holding
host, port = address
```

Хотя кортеж поддерживает большинство операций списков (индексирование, сегментацию и конкатенацию), его элементы не могут изменяться после создания. Другими словами, вы не сможете заменить, удалить или присоединить новые элементы к существующему кортежу. Его лучше рассматривать как один неизменяемый объект из нескольких частей, а не как набор отдельных элементов (список).

Кортежи и списки часто используются вместе для представления данных. Следующая программа показывает, как можно читать файл со столбцами данных, разделенными запятыми:

```
# Файл со строками в формате ``название,количество,цена"
filename = 'portfolio.csv'

portfolio = []
with open(filename) as file:
    for line in file:
        row = line.split(',')
        name = row[0]
        shares = int(row[1])
        price = float(row[2])
        holding = (name, shares, price)
        portfolio.append(holding)
```

Созданный этой программой список похож на двумерный массив со строками и столбцами. Каждая строка представляется кортежем, а обращение к ней может выглядеть так:

```
>>> portfolio[0]
('AA', 100, 32.2)
>>> portfolio[1]
('IBM', 50, 91.1)
>>>
```

Можно обращаться и к отдельным элементам данных:

```
>>> portfolio[1][1]
50
>>> portfolio[1][2]
91.1
>>>
```

Перебор всех записей и распаковка полей в набор переменных происходят так:

```
total = 0.0
for name, shares, price in portfolio:
    total += shares * price
```

Вместо этого также можно воспользоваться списковым включением:

```
total = sum([shares * price for _, shares, price in portfolio])
```

При переборе данных кортежа переменная `_` может использоваться для обозначения игнорируемого значения. В нашем примере это значит, что игнорироваться будет первый элемент (название).

1.10. МНОЖЕСТВА

Множество — это неупорядоченный набор уникальных объектов. Оно используется для поиска неповторяющихся значений или решения проблем, связанных с принадлежностью. Для создания множества заключите коллекцию значений в фигурные скобки или передайте уже созданную коллекцию элементов при вызове `set()`:

```
names1 = { 'IBM', 'MSFT', 'AA' }
names2 = set(['IBM', 'MSFT', 'HPE', 'IBM', 'CAT'])
```

Элементами множеств обычно могут быть только неизменяемые объекты. Можно создать множество чисел, строк или кортежей, но не множество списков. Но многие популярные объекты будут работать во множествах — если сомневаетесь, попробуйте.

В отличие от списков и кортежей, элементы множеств не упорядочены и не могут индексироваться числами. Еще во множестве не может быть повторяющихся элементов. Например, при проверке значения `names2` из фрагмента выше вы получите следующий результат:

```
>>> names2
{'CAT', 'IBM', 'MSFT', 'HPE'}
>>>
```

Обратите внимание, что строка 'IBM' встречается только единожды. Порядок элементов непредсказуем. Ввод на вашем компьютере может отличаться от показанного. Порядок может даже изменяться между разными запусками интерпретатора на одной машине.

При работе с существующими данными вы можете использовать для создания множества генератор множеств. Например, следующая команда преобразует во множество все названия акций в данных из прошлого раздела:

```
names = { s[0] for s in portfolio }
```

Для создания пустого множества используйте вызов `set()` без аргументов:

```
r = set() # Initially empty set
```

Множества поддерживают стандартный набор операций, включая объединение, пересечение, разность и симметричную разность:

```
a = t | s # Объединение {'MSFT', 'CAT', 'HPE', 'AA', 'IBM'}
b = t & s # Пересечение {'IBM', 'MSFT'}
c = t - s # Разность {'CAT', 'HPE'}
d = s - t # Разность {'AA'}
e = t ^ s # Симметричная разность {'CAT', 'HPE', 'AA'}
```

Операция разности `s - t` возвращает элементы `s`, не входящие в `t`. Симметричная разность `s ^ t` возвращает элементы, входящие либо в `s`, либо в `t`, но не в оба множества сразу.

Новые элементы могут добавляться во множество методом `add()` или `update()`:

```
t.add('DIS') # Добавление одного элемента
s.update({'JJ', 'GE', 'ACME'}) # Добавление нескольких элементов в s
```

Элементы удаляются вызовом `remove()` или `discard()`:

```
t.remove('IBM') # Удаляет 'IBM' или выдает ошибку KeyError,
                # если элемент отсутствует.
s.discard('SCOX') # Удаляет элемент 'SCOX', если он существует.
```

Разница методов `remove()` и `discard()` в том, что `discard()` не выдает исключение при отсутствии элемента.

1.11. СЛОВАРИ

Словарь (dictionary) определяет соответствие между ключами и значениями. Для создания словаря заключите пары «ключ — значение», разделенные двоеточием, в фигурные скобки (`{}`):

```
s = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10
}
```

Для обращения к компонентам словаря применяется оператор индексирования:

```
name = s['name']
cost = s['shares'] * s['price']
```

Операции вставки и изменения объектов работают так:

```
s['shares'] = 75
s['date'] = '2007-06-07'
```

Словарь — удобный способ определить объект, состоящий из именованных полей. Но он часто используется и как определение отображений для быстрого поиска по неупорядоченным данным. Например, следующий словарь содержит информацию о ценах на акции:

```
prices = {
    'GOOG' : 490.1,
    'AAPL' : 123.5,
    'IBM' : 91.5,
    'MSFT' : 52.13
}
```

С таким словарем можно узнать цену по названию акций:

```
p = prices['IBM']
```

Наличие элемента в словаре проверяется оператором `in`:

```
if 'IBM' in prices:
    p = prices['IBM']
else:
    p = 0.0
```

Эту последовательность шагов можно выполнить более компактно с использованием метода `get()`:

```
p = prices.get('IBM', 0.0) # prices['IBM'] if it exists, else 0.0
```

Для удаления элемента словаря используется команда `del`:

```
del prices['GOOG']
```


Строки — самая распространенная разновидность ключей. Но вы можете использовать любые другие объекты Python, включая числа и кортежи. Например, кортежи часто применяются для формирования составных ключей:

```
prices = { }
prices[('IBM', '2015-02-03')] = 91.23
prices['IBM', '2015-02-04'] = 91.42 # Без круглых скобок
```

В ключах могут храниться любые объекты, в том числе и другие словари. Но изменяемые структуры данных (списки, множества и словари) не могут быть ключами.

Словари часто используются как структурные элементы для разных алгоритмов и задач обработки данных. Одна из таких задач — формирование табличного представления данных. Вот как можно подсчитать общее число акций для каждого названия из вышеупомянутых данных:

```
portfolio = [
    ('ACME', 50, 92.34),
    ('IBM', 75, 102.25),
    ('PHP', 40, 74.50),
    ('IBM', 50, 124.75)
]

total_shares = { s[0]: 0 for s in portfolio }
for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = {'IBM': 125, 'ACME': 50, 'PHP': 40}
```

Здесь `{ s[0]: 0 for s in portfolio }` — пример словарного включения. Эта форма создает словарь пар «ключ — значение» из другой коллекции данных. В этом случае она создает исходный словарь, связывающий названия акций с 0. Следующий цикл `for` перебирает элементы словаря и суммирует все акции для каждого биржевого сокращения.

Многие распространенные задачи обработки данных были реализованы библиотечными модулями. Например, модуль `collections` содержит объект `Counter`, которым можно воспользоваться в такой задаче:

```
from collections import Counter

total_shares = Counter()
for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = Counter({'IBM': 125, 'ACME': 50, 'PHP': 40})
```

Пустой словарь создается одним из двух способов:

```
prices = {}      # Пустой словарь
prices = dict() # Пустой словарь
```

Использование `{}` для создания пустых списков более приемлемо. Но оно требует внимательности, ведь может показаться, что вы пытаетесь создать пустое множество (во избежание путаницы используйте `set()`); `dict()` обычно применяется для создания словарей по парам «ключ — значение»). Пример:

```
pairs = [('IBM', 125), ('ACME', 50), ('PHP', 40)]
d = dict(pairs)
```

Чтобы получить список ключей словаря, преобразуйте словарь в список:

```
syms = list(prices) # syms = ['AAPL', 'MSFT', 'IBM', 'GOOG']
```

Для получения ключей можно добавить и метод `dict.keys()`:

```
syms = prices.keys()
```

Разница этих способов в том, что `keys()` возвращает специальное «представление ключей», присоединенное к словарю и активно отражающее изменения в нем. Пример:

```
>>> d = { 'x': 2, 'y':3 }
>>> k = d.keys()
>>> k
dict_keys(['x', 'y'])
>>> d['z'] = 4
>>> k
dict_keys(['x', 'y', 'z'])
>>>
```

Ключи всегда следуют в том порядке, в котором элементы изначально вставлялись в словарь. Списковое включение выше сохраняет такой порядок. Это будет удобно при использовании словарей для представления данных «ключ — значение», читаемых из файлов и других источников. Словарь сохраняет порядок ввода, что упрощает чтение программы и ее отладку. Это пригодится и в том случае, если вы захотите записать данные обратно в файл. С другой стороны, до выхода версии Python 3.6 такой порядок не был гарантирован. Поэтому вы не можете полагаться на него при совмещении со старыми версиями Python. Порядок не гарантируется и при выполнении множественных операций удаления и вставки.

Для получения значений из словаря используется метод `dict.values()`. Чтобы получить пары «ключ — значение», применяйте метод `dict.items()`.

Например, перебор всего содержимого словаря по парам «ключ — значение» выполняется так:

```
for sym, price in prices.items():
    print(f'{sym} = {price}')
```

1.12. ПЕРЕБОР И ЦИКЛЫ

Из всех конструкций циклов чаще всего используется команда `for` для перебора коллекций. Одна из распространенных форм перебирает все элементы последовательности (строки, списка или кортежа). Например:

```
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(f'2 to the {n} power is {2**n}')
```

Здесь переменной `n` последовательно присваиваются элементы списка `[1, 2, 3, 4, ..., 9]`. Перебор по диапазонам целых чисел встречается довольно часто, поэтому для него предусмотрена специальная сокращенная запись:

```
for n in range(1, 10):
    print(f'2 to the {n} power is {2**n}')
```

Функция `range(i, j [,step])` создает объект, представляющий диапазон целых чисел со значениями от `i` до `j` (не включая последнее). Если начальное значение опущено, оно считается равным нулю. В третьем аргументе может передаваться необязательное приращение. Несколько примеров:

```
a = range(5)           # a = 0, 1, 2, 3, 4
b = range(1, 8)        # b = 1, 2, 3, 4, 5, 6, 7
c = range(0, 14, 3)    # c = 0, 3, 6, 9, 12
d = range(8, 1, -1)    # d = 8, 7, 6, 5, 4, 3, 2
```

Объект, созданный вызовом `range()`, вычисляет представляемые им значения по запросу. Так функция эффективно работает даже с большими диапазонами чисел.

Команда `for` не ограничивается последовательностями целых чисел. С ее помощью можно перебирать разные объекты, включая строки, списки, словари и файлы. Пример:

```
message = 'Hello World'
# Вывод отдельных символов из message
for c in message:
    print(c)

names = ['Dave', 'Mark', 'Ann', 'Phil']
# Вывод элементов списка
```

```

for name in names:
    print(name)

prices = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# Вывод всех элементов словаря
for key in prices:
    print(key, '=', prices[key])

# Вывод всех строк файла
with open('foo.txt') as file:
    for line in file:
        print(line, end='')

```

Цикл `for` относится к числу самых мощных возможностей языка Python. Вы можете создавать свои объекты-итераторы и функции-генераторы, поставляющие циклу последовательности значений. Дополнительную информацию об итераторах и генераторах вы найдете в главе 6.

1.13. ФУНКЦИИ

Для определения функций используется команда `def`:

```

def remainder(a, b):
    q = a // b      # // целочисленное деление
    r = a - q * b
    return r

```

Чтобы вызвать функцию, укажите ее имя, за которым следуют аргументы в круглых скобках. Например, `result = remainder(37, 15)`.

При написании функций программисты обычно включают в начало строку документации. Она дает информацию для команды `help()` и может использоваться IDE и другими средствами разработки для помощи программисту:

```

def remainder(a, b):
    """
    Вычисляет остаток от деления a на b
    """
    q = a // b
    r = a - q * b
    return r

```

Если входные и выходные данные функции не очевидны по ее имени, их можно снабдить аннотациями типов:

```

def remainder(a: int, b: int) -> int:
    """
    Вычисляет остаток от деления a на b
    """

```

```
...
q = a // b
r = a - q * b
return r
```

Такие аннотации создаются только для предоставления информации, во время выполнения они не соблюдаются. Ничто не мешает вызвать функцию выше с нецелыми значениями, например `result = remainder(37.5, 3.2)`.

Для возврата нескольких значений из функции используйте кортеж:

```
def divide(a, b):
    q = a // b      # Если a и b целые числа, то q — целое число
    r = a - q * b
    return (q, r)
```

Набор значений, возвращаемых в виде кортежа, можно распаковать на отдельные переменные:

```
quotient, remainder = divide(1456, 33)
```

Чтобы присвоить параметру функции значение по умолчанию, используйте команду присваивания:

```
def connect(hostname, port, timeout=300):
    # Тело функции
    ...
```

Если в определении функции заданы значения по умолчанию, эти параметры можно опустить при следующих вызовах функции. Отсутствующим аргументам присваиваются значения по умолчанию.

Пример:

```
connect('www.python.org', 80)
connect('www.python.org', 80, 500)
```

Аргументы по умолчанию часто используются для дополнительных функций. Если таких аргументов будет много, это затруднит чтение программы. Поэтому в таких случаях рекомендуется использовать ключевые:

```
connect('www.python.org', 80, timeout=500)
```

Если имена аргументов известны, все они могут указываться при вызове функции. Порядок, в котором они перечислены, не имеет значения. Например, следующий вызов вполне допустим:

```
connect(port=80, hostname='www.python.org')
```

Переменные, созданные внутри функции, обладают локальной областью видимости. Другими словами, переменная определяется только внутри тела функции и уничтожается при возвращении из нее. Функция может обращаться и к переменным, определенным за ее пределами, — при условии, что они определяются в том же файле:

```
debug = True           # Глобальная переменная

def read_data(filename):
    if debug:
        print('Reading', filename)
    ...
```

Правила видимости подробнее описаны в главе 5.

1.14. ИСКЛЮЧЕНИЯ

Если в программе происходит ошибка, она выдает исключение и выводит сообщение с трассировкой:

```
Traceback (most recent call last):
  File "readport.py", line 9, in <module>
    shares = int(row[1])
ValueError: invalid literal for int() with base 10: 'N/A'
```

В сообщении указан тип ошибки и место ее появления. Обычно ошибка приводит к завершению программы. Но исключения можно перехватывать и обрабатывать командами `try` и `except`:

```
portfolio = []
with open('portfolio.csv') as file:
    for line in file:
        row = line.split(',')
        try:
            name = row[0]
            shares = int(row[1])
            price = float(row[2])
            holding = (name, shares, price)
            portfolio.append(holding)
        except ValueError as err:
            print('Bad row:', row)
            print('Reason:', err)
```

В этом коде при возникновении исключения `ValueError` подробная информация о причине ошибки помещается в переменные `err` и `control`, передаваемые коду в блоке `except`. При исключениях другого типа программа

аварийно завершается, как и прежде. При отсутствии ошибки код в блоке `except` игнорируется. Если исключение обработано, выполнение программы продолжается с команды, следующей за последним блоком `except`. Программа не возвращает управление в точку, где произошло исключение.

Команда `raise` выдает исключение в программе. Ей необходимо передать имя этого исключения. В следующем примере выдается встроенное исключение `RuntimeError`:

```
raise RuntimeError('Computer says no')
```

Обработка исключений часто усложняет управление системными ресурсами (блокировками, файлами, сетевыми подключениями и т. д.). В некоторых ситуациях есть действия, которые должны выполняться в любом случае. Для этого используется команда `try-finally`. Вот пример блокировки, которую нужно снять для предотвращения взаимоблокировки (deadlock):

```
import threading
lock = threading.Lock()
...
lock.acquire()
# Если блокировка была захвачена, она ДОЛЖНА быть снята
try:
    ...
    команды
    ...
finally:
    lock.release() # Выполняется всегда
```

Для упрощения такого стиля программирования многие объекты, связанные с управлением ресурсами, поддерживают команду `with`. Измененная версия этого кода выглядит так:

```
with lock:
    ...
    команды
    ...
```

В этом примере объект `lock` автоматически захватывается при выполнении команды `with`. Когда выполнение выходит за пределы контекста блока `with`, блокировка автоматически снимается. Это происходит независимо от того, что происходит внутри блока `with`. Например, если возникает исключение, блокировка снимается, когда управление покидает контекст блока.

Команда `with` обычно совместима только с объектами, связанными с системными ресурсами или исполнительной средой, — файлами, подключениями,

блокировками и т. д. Но объекты, определяемые пользователем, могут реализовать свою обработку, как описано в главе 3.

1.15. ЗАВЕРШЕНИЕ ПРОГРАММЫ

Программа завершается, когда в ней не остается команд для выполнения, или при выдаче неперехваченного исключения `SystemExit`. Принудительно завершить программу можно так:

```
raise SystemExit()           # Выход без сообщения об ошибке
raise SystemExit("Something is wrong") # Выход с ошибкой
```

При выходе интерпретатор делает все возможное для очистки всех активных объектов от мусора. Но если вам нужно выполнить конкретное завершающее действие (удалить файлы, закрыть сетевое подключение), зарегистрируйте его в модуле `atexit`:

```
import atexit

# Пример
connection = open_connection("deaddot.com")

def cleanup():
    print "Going away..."
    close_connection(connection)

atexit.register(cleanup)
```

1.16. ОБЪЕКТЫ И КЛАССЫ

Все используемые в программах значения являются объектами. Объект состоит из внутренних данных и методов, выполняющих разные операции с этими данными. Вы уже использовали объекты и методы при работе с такими встроенными типами, как строки и списки. Пример:

```
items = [37, 42] # Создание объекта списка
items.append(73) # Вызов метода append()
```

Функция `dir()` выводит список методов, доступных для объекта. Это полезный инструмент для интерактивных экспериментов при отсутствии IDE. Пример:

```
>>> items = [37, 42]
>>> dir(items)
```



```
[ '__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
...
'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort' ]
>>>
```

В списке методов объектов вы увидите такие знакомые методы, как `append()` и `insert()`. Но встречаются и специальные методы, имена которых начинаются и заканчиваются двойным символом подчеркивания. Они реализуют разные операторы. Например, метод `__add__()` используется для реализации `+`. Подробнее об этих методах вы узнаете в следующих главах.

```
>>> items.__add__([73, 101])
[37, 42, 73, 101]
>>>
```

Команда `class` предназначена для определения новых типов объектов и для объектно-ориентированного программирования. Например, следующий класс определяет стек с операциями `push()` и `pop()`:

```
class Stack:
    def __init__(self): # Инициализация стека
        self._items = [ ]

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __repr__(self):
        return f'<{type(self).__name__} at 0x{id(self):x},
size={len(self)}>'

    def __len__(self):
        return len(self._items)
```

Внутри определения класса методы обозначаются командой `def`. Первый аргумент каждого метода всегда обозначает сам объект. По общепринятому соглашению этому аргументу присваивается имя `self`. Все операции, где задействованы атрибуты объекта, должны явно ссылаться на переменную `self`.

Методы, имена которых начинаются и заканчиваются двумя символами подчеркивания, выполняют специальные операции. Например, метод `__init__` используется для инициализации объекта. В нашем случае `__init__` создает внутренний список для хранения данных стека.

Код использования класса может выглядеть примерно так:

```
s = Stack()           # Создание стека
s.push('Dave')        # Занесение данных в стек
s.push(42)
s.push([3, 4, 5])
x = s.pop()           # В x сохраняется [3,4,5]
y = s.pop()           # В y сохраняется 42
```

Возможно, вы заметили, что внутри класса в методах используется внутренняя переменная `_items`. В Python нет механизмов сокрытия или защиты данных. Но у программистов есть соглашение: имена, начинающиеся с одного символа подчеркивания, считаются «приватными». В этом примере переменную `_items` нужно рассматривать как внутреннюю реализацию и не использовать за пределами самого класса `Stack`. Учтите, что выполнение этого соглашения нигде не поддерживается. Обратиться к `_items` можно в любой момент. Придется только ответить на вопросы ваших коллег во время рецензирования вашего кода.

Методы `__repr__()` и `__len__()` определяются для лучшего взаимодействия объекта со средой. Здесь `__len__()` обеспечивает работу `Stack` со встроенной функцией `len()`, а `__repr__()` определяет, в каком формате должны выводиться объекты `Stack`. Метод `__repr__()` желательно определять всегда, так как он упрощает отладку.

```
>>> s = Stack()
>>> s.push('Dave')
>>> s.push(42)
>>> len(s)
2
>>> s
<Stack at 0x10108c1d0, size=2>
>>>
```

Самое главное свойство объектов в том, что вы можете расширять или переопределять функциональность существующих классов через механизм наследования. Допустим, вы хотите добавить метод, который будет менять местами два верхних элемента в стеке. Для этого можно написать новый класс:

```
class MyStack(Stack):
    def swap(self):
        a = self.pop()
        b = self.pop()
        self.push(a)
        self.push(b)
```

`MyStack` идентичен `Stack`, не считая того, что он определяет новый метод `swap()`.

```
>>> s = MyStack()
>>> s.push('Dave')
>>> s.push(42)
>>> s.swap()
>>> s.pop()
'Dave'
>>> s.pop()
42
>>>
```

Наследование может использоваться и для изменения поведения существующего метода. Предположим, вы хотите ограничить стек так, чтобы он мог использоваться для хранения только числовых данных. Напишите такой класс:

```
class NumericStack(Stack):
    def push(self, item):
        if not isinstance(item, (int, float)):
            raise TypeError('Expected an int or float')
        super().push(item)
```

Здесь `push()` был переопределен с добавлением дополнительной проверки. Операция `super()` используется для вызова предыдущего определения `push()`. Пример использования этого класса:

```
>>> s = NumericStack()
>>> s.push(42)
>>> s.push('Dave')
Traceback (most recent call last):
...
TypeError: Expected an int or float
>>>
```

Часто наследование оказывается не лучшим решением. Допустим, вы хотите определить на базе стека простой калькулятор с четырьмя функциями, который работает примерно так:

```
>>> # Вычисление 2 + 3 * 4
>>> calc = Calculator()
>>> calc.push(2)
>>> calc.push(3)
>>> calc.push(4)
>>> calc.mul()
>>> calc.add()
>>> calc.pop()
14
>>>
```

Взглянув на этот код, можно увидеть вызовы `push()` и `pop()` и решить, что класс `Calculator` можно определить наследованием от `Stack`. Хотя такое решение будет работать, все же лучше определить `Calculator` как совершенно отдельный класс:

```
class Calculator:
    def __init__(self):
        self._stack = Stack()

    def push(self, item):
        self._stack.push(item)

    def pop(self):
        return self._stack.pop()

    def add(self):
        self.push(self.pop() + self.pop())

    def mul(self):
        self.push(self.pop() * self.pop())

    def sub(self):
        right = self.pop()
        self.push(self.pop() - right)

    def div(self):
        right = self.pop()
        self.push(self.pop() / right)
```

В этой реализации `Calculator` содержит `Stack` как часть внутренней реализации. Здесь мы видим применение механизма *композиции*. Методы `push()` и `pop()` делегируют обязанности внутреннему экземпляру `Stack`. Главная причина выбора этого подхода в том, что калькулятор вряд ли можно рассматривать как разновидность стека. Это отдельная концепция — другая разновидность объекта. Можно привести и другую аналогию: в вашем телефоне установлен процессор, но обычно вы не рассматриваете телефон как разновидность процессора.

1.17. МОДУЛИ

По мере того как ваши программы увеличиваются в размерах, вы захотите разбить их на несколько файлов, чтобы облегчить обслуживание. Для этого используется команда `import`. Чтобы создать модуль, сохраните соответствующие команды и определения в файле с суффиксом `.py` и с таким же именем, как у модуля:

```
# readport.py
#
# Reads a file of 'NAME,SHARES,PRICE' data

def read_portfolio(filename):
    portfolio = []
    with open(filename) as file:
        for line in file:
            row = line.split(',')
            try:
                name = row[0]
                shares = int(row[1])
                price = float(row[2])
                holding = (name, shares, price)
                portfolio.append(holding)
            except ValueError as err:
                print('Bad row:', row)
                print('Reason:', err)
    return portfolio
```

Чтобы использовать модуль в других файлах, введите команду `import`. Пример модуля `pcost.py`, где используется функция `read_portfolio()`:

```
# pcost.py

import readport

def portfolio_cost(filename):
    '''
    Вычислить общую стоимость портфеля shares*price
    '''
    port = readport.read_portfolio(filename)
    return sum(shares * price for _, shares, price in port)
```

Команда `import` создает новое пространство имен (или среду) и выполняет все команды в связанном файле `.py` из этого пространства имен. Чтобы обратиться к содержимому пространства имен после импортирования модуля, используйте имя модуля как префикс, как в вызове `readport.read_portfolio()` в прошлом примере.

Если команда `import` завершается ошибкой с исключением `ImportError`, проверьте несколько аспектов вашей среды. Сначала убедитесь, что файл с именем `readport.py` существует. Затем проверьте каталоги, входящие в `sys.path`. Если ваш файл не был сохранен в одном из них, Python не найдет его.

Если вы хотите импортировать модуль под другим именем, укажите команду `import` с необязательным квалификатором `as`:

```
import readport as rp
port = rp.read_portfolio('portfolio.dat')
```

Для импорта конкретных определений в текущее пространство имен используйте команду `from`:

```
from readport import read_portfolio
port = read_portfolio('portfolio.dat')
```

По аналогии с объектами функция `dir()` выводит содержимое модуля. Это полезный инструмент для экспериментов в интерактивном режиме.

```
>>> import readport
>>> dir(readport)
['_builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'read_portfolio']
...
>>>
```

У Python большая стандартная библиотека модулей, которая упрощает решение некоторых задач программирования. Например, модуль `csv` предназначен для работы с файлами, содержащими данные в формате CSV (значения, разделенные запятыми).

Вы можете использовать его в своей программе так:

```
# readport.py
#
# Читает файл с данными 'НАЗВАНИЕ,КОЛИЧЕСТВО,ЦЕНА'

import csv

def read_portfolio(filename):
    portfolio = []
    with open(filename) as file:
        rows = csv.reader(file)
        for row in rows:
            try:
                name = row[0]
                shares = int(row[1])
                price = float(row[2])
                holding = (name, shares, price)
                portfolio.append(holding)
            except ValueError as err:
                print('Bad row:', row)
                print('Reason:', err)
    return portfolio
```

В Python есть и множество сторонних модулей, которые можно установить для решения практически любых задач (включая чтение файлов CSV). См. <https://pypi.org>.

1.18. НАПИСАНИЕ СЦЕНАРИЕВ

Любой файл может выполняться либо как сценарий, либо как библиотека, импортированная командой `import`. Для улучшения поддержки импортирования код сценария часто заключается в блок, где проверяется имя модуля:

```
# readport.py
#
# Читает файл с данными 'НАЗВАНИЕ,КОЛИЧЕСТВО,ЦЕНА'

import csv

def read_portfolio(filename):
    ...

def main():
    portfolio = read_portfolio('portfolio.csv')
    for name, shares, price in portfolio:
        print(f'{name:>10s} {shares:10d} {price:10.2f}')

if __name__ == '__main__':
    main()
```

Встроенная переменная `__name__` всегда содержит имя вмещающего модуля. Если программа выполняется как основной сценарий (например, командой `python readport.py`), переменной `__name__` присваивается `'__main__'`. Если же код импортируется командой `import readport`, то переменной `__name__` будет присвоено значение `'readport'`.

Как показано выше, в программе жестко зафиксировано имя файла `'portfolio.csv'`. Возможны и другие решения. Например, имя файла может вводиться пользователем или передаваться в аргументе командной строки. Для этого используйте встроенную функцию `input()` или список `sys.argv`. Измененная версия функции `main()` может выглядеть так:

```
def main(argv):
    if len(argv) == 1:
        filename = input('Enter filename: ')
    elif len(argv) == 2:
        filename = argv[1]
```

```

else:
    raise SystemExit(f'Usage: {argv[0]} [ filename ]')
portfolio = read_portfolio(filename)
for name, shares, price in portfolio:
    print(f'{name:>10s} {shares:10d} {price:10.2f}')

if __name__ == '__main__':
    import sys
    main(sys.argv)

```

Эту программу можно запустить из командной строки двумя способами:

```

bash % python readport.py
Enter filename: portfolio.csv
...
bash % python readport.py portfolio.csv
...
bash % python readport.py a b c
Usage: readport.py [ filename ]
bash %

```

Для очень простых программ часто бывает достаточно такой обработки аргументов `sys.argv`. Если вам понадобится что-то посложнее, используйте модуль стандартной библиотеки `argparse`.

1.19. ПАКЕТЫ

В больших программах код часто разбивается на пакеты. Пакет — это иерархический набор модулей. В файловой системе код распределяется по файлам в каталоге:

```

tutorial/
  __init__.py
  readport.py
  pcost.py
  stack.py
  ...

```

Каталог должен содержать файл `__init__.py`, который может быть пустым. После создания файлов вы сможете использовать вложенные команды импорта:

```

import tutorial.readport

port = tutorial.readport.read_portfolio('portfolio.dat')

```


Если вам не нравятся длинные имена, их можно сократить следующей командой импорта:

```
from tutorial.readport import read_portfolio

port = read_portfolio('portfolio.dat')
```

При работе с пакетами иногда возникает неочевидная проблема импорта между файлами одного пакета. В более раннем примере был приведен модуль `pcost.py`, который начинался с такой команды импорта:

```
# pcost.py
import readport
...
```

Если файлы `pcost.py` и `readport.py` будут перемещены в пакет, команда `import` перестанет работать. Для исправления ошибки можно использовать полную команду импорта модуля:

```
# pcost.py
from tutorial import readport
...
```

Также можно использовать команду импорта, относительную для пакета:

```
# pcost.py
from . import readport
...
```

К достоинствам последней формы можно отнести то, что она не фиксирует имя пакета. Это позволяет позднее легко переименовать его или перемещать в вашем проекте.

Другие подробности, связанные с использованием пакетов, рассматриваются в главе 8.

1.20. СТРУКТУРА ПРИЛОЖЕНИЯ

Когда вы начнете писать больше кода на Python, вы, возможно, будете работать и над более крупными приложениями, содержащими как ваш собственный код, так и сторонние зависимости. Управление зависимостями — сложная область, которая продолжает развиваться. Мнения по поводу того, что именно можно считать передовыми практиками, тоже расходятся.

Но есть ряд основных моментов, связанных со структурой приложения, о которых важно знать.

Во-первых, большие кодовые базы принято упорядочивать с созданием пакетов (то есть каталогов с файлами `.py`, содержащих специальный файл `__init__.py`). При этом каталогу верхнего уровня присваивается уникальное имя пакета. Каталог пакета прежде всего предназначен для управления командами `import` и пространствами имен модулей, используемыми в программировании. Ваш код должен быть изолирован от остального кода.

Помимо основного исходного кода, в проекте могут содержаться тесты, примеры, сценарии и документация. Эти дополнительные материалы обычно размещаются в специальных каталогах отдельно от исходного кода. Так, в проекте принято создавать каталог верхнего уровня для всего проекта и размещать в нем всю текущую работу. Типичная структура проекта может выглядеть так:

```
tutorial-project/
  tutorial/
    __init__.py
    readport.py
    pcost.py
    stack.py
    ...
  tests/
    test_stack.py
    test_pcost.py
    ...
  examples/
    sample.py
    ...
  doc/
    tutorial.txt
    ...
```

Помните, что единственно правильного варианта организации проекта нет. Природа вашей задачи может диктовать другую структуру. Но пока основной набор файлов с исходным кодом находится в соответствующем пакете (еще раз: каталоге с файлом `__init__.py`), проблем быть не должно.

1.21. УПРАВЛЕНИЕ СТОРОННИМИ ПАКЕТАМИ

Для Python есть большая библиотека внешних пакетов, доступных на сайте Python Package Index (<https://pypi.org>). Возможно, некоторые из них будут

использоваться в вашем коде. Для установки стороннего пакета используйте менеджер пакетов — например, `pip`:

```
bash % python3 -m pip install somepackage
```

Установленные пакеты размещаются в специальном каталоге `site-packages`, который можно найти при анализе значения `sys.path`. Например, на UNIX пакеты могут храниться в каталоге `/usr/local/lib/python3.8/site-packages`. Если у вас возникнут вопросы относительно того, откуда взялся пакет, импортируйте его в интерпретаторе и проверьте атрибут `__file__`:

```
>>> import pandas
>>> pandas.__file__
'/usr/local/lib/python3.8/site-packages/pandas/__init__.py'
>>>
```

Одна из потенциальных проблем с установкой пакетов в том, что у вас может не быть разрешений для изменения локально установленной версии Python. Но даже если они есть, делать это все равно не рекомендуется. Например, во многих системах уже установлена копия Python, которая используется разными системными утилитами. Изменение установки этой версии Python часто приводит к нежелательным последствиям.

Чтобы создать «песочницу», где можно устанавливать пакеты и работать с ними, не боясь что-то сломать, создайте *виртуальную среду* такой командой:

```
bash % python3 -m venv myproject
```

Эта команда создает специальную установку Python в каталоге `myproject/`. В нем вы найдете исполняемый файл интерпретатора и библиотеку для безопасной установки пакетов. Например, запустив `myproject/bin/python3`, вы получите интерпретатор, настроенный для вашего личного использования. Вы можете устанавливать пакеты для этого интерпретатора, не беспокоясь, что это нарушит какую-либо часть установки Python по умолчанию.

Для установки пакета используйте команду `pip`, как и прежде. Но не забудьте указать верный интерпретатор:

```
bash % ./myproject/bin/python3 -m pip install somepackage
```

Есть много программных средств, упрощающих использование команд `pip` и `venv`. Кроме того, IDE может взять на себя часть ваших хлопот. Так как эта часть Python постоянно изменяется, больше никаких советов здесь не дается.

1.22. PYTHON ПОДСТРАИВАЕТСЯ ПОД ВАШИ ЗАПРОСЫ

На заре существования Python часто приходилось слышать девиз «Он подстраивается под ваши запросы». Даже сегодня Python — маленький язык программирования с полезной подборкой встроенных объектов: списков, множеств и словарей. Для решения широкого круга практических задач не нужно ничего, кроме базовых знаний из этой главы. Делая первые шаги на пути изучения Python, помните: хотя всегда есть более сложные способы решения задачи, может быть и простой способ с использованием базовых возможностей Python. Если у вас возникнут сомнения, выбирайте простое решение — в будущем вы не пожалеете об этом.

ГЛАВА 2

Операторы, выражения и обработка данных

В этой главе описаны выражения, операторы и правила вычисления в языке Python, относящиеся к обработке данных. Выражения лежат в основе выполнения полезных вычислений. Более того, сторонние библиотеки могут изменять поведение Python для удобства пользователя. Здесь выражения рассматриваются на высоком уровне. В главе 3 описаны базовые протоколы, которые могут использоваться для настройки поведения интерпретатора.

2.1. ЛИТЕРАЛЫ

Литерал — это значение, определяемое в программе, например 42, 4.2 или 'forty-two'.

Целочисленные литералы — это целые значения произвольного размера со знаком. Целые числа тоже могут задаваться в двоичной, восьмеричной или шестнадцатеричной системе:

```
42      # Десятичный литерал
0b101010 # Двоичный литерал
0o52    # Восьмеричный литерал
0x2a    # Шестнадцатеричный литерал
```

Основание не сохраняется вместе со значением целого числа. Все приведенные выше литералы будут выводиться в виде 42. Для преобразования целого числа в строку, представляющую его значение в разных системах счисления, используйте встроенные функции `bin(x)`, `oct(x)` или `hex(x)`.

Числа с плавающей точкой можно записать, добавив десятичную точку или используя экспоненциальное представление, где *e* или *E* указывает показатель степени. Все следующие формы — числа с плавающей точкой:

```

4.2
42.
.42
4.2e+2
4.2E2
-4.2e-2

```

Во внутреннем представлении числа с плавающей точкой хранятся в виде значений IEEE 754 с двойной точностью (64-битных).

В числовых литералах одиночный символ подчеркивания (`_`) может использоваться для визуального разделения цифр:

```

123_456_789
0x1234_5678
0b111_00_101
123.789_012

```

Разделитель не сохраняется как часть числа — он только упрощает чтение больших числовых литералов в исходном коде.

Логические литералы записываются в виде `True` и `False`.

При записи строковых литералов символы заключаются в одинарные, двойные или тройные кавычки.

Строки в одинарных и двойных кавычках должны записываться в одной строке программы. Строки в тройных кавычках могут охватывать несколько строк:

```

'hello world'
"hello world"
'''hello world'''
"""hello world"""

```

Литералы кортежей, списков, множеств и словарей записываются так:

```

(1, 2, 3)           # Кортеж
[1, 2, 3]           # Список
{1, 2, 3}           # Множество
{'x':1, 'y':2, 'z':3} # Словарь

```

2.2. ВЫРАЖЕНИЯ И АДРЕСА ПАМЯТИ

Выражение — некое вычисление, в результате которого получается конкретное значение. Это сочетание литералов, имен, операторов, вызовов функций

и методов. Выражение всегда может находиться в правой части команды присваивания, может использоваться как операнд в операциях в других выражениях или же передаваться как аргумент функции. Например:

```
value = 2 + 3 * 5 + sqrt(6+7)
```

Такие операторы, как `+` (сложение) или `*` (умножение), выполняют операцию с объектами, указанными как операнды. `sqrt()` — функция, которая применяется к входным аргументам.

Левая часть присваивания означает место, где хранится ссылка на объект. Ячейкой памяти может быть простой идентификатор, например `value`. Ею может быть и атрибут объекта или индекс в контейнере:

```
a = 4 + 2
b[1] = 4 + 2
c['key'] = 4 + 2
d.value = 4 + 2
```

Чтение значения из места ячейки памяти — тоже выражение. Пример:

```
value = a + b[1] + c['key']
```

Присваивание значения и вычисление выражения — разные концепции. Оператор присваивания не может быть частью выражения:

```
while line=file.readline(): # Синтаксическая ошибка
    print(line)
```

Но оператор «выражения с присваиванием» (`:=`) может использоваться для выполнения объединенного действия вычисления выражения и присваивания. Пример:

```
while (line:=file.readline()):
    print(line)
```

Оператор `:=` обычно применяется в таких командах, как `if` и `while`. Попытка использования его в качестве обычного оператора присваивания приводит к синтаксической ошибке, если только вы не заключите его в круглые скобки.

2.3. СТАНДАРТНЫЕ ОПЕРАТОРЫ

Вы можете сделать так, чтобы объекты Python работали с любыми операторами из табл. 2.1.

Таблица 2.1. Стандартные операторы

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|-----------------------------------|--------------------------------------|
| <code>x + y</code> | Сложение |
| <code>x - y</code> | Вычитание |
| <code>x * y</code> | Умножение |
| <code>x / y</code> | Деление |
| <code>x // y</code> | Целочисленное деление |
| <code>x @ y</code> | Умножение матриц |
| <code>x ** y</code> | Возведение в степень (x в степень y) |
| <code>x % y</code> | Остаток (от деления x на y) |
| <code>x << y</code> | Сдвиг влево |
| <code>x >> y</code> | Сдвиг вправо |
| <code>x & y</code> | Битовая операция И |
| <code>x y</code> | Битовая операция ИЛИ |
| <code>x ^ y</code> | Битовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ |
| <code>~x</code> | Битовое отрицание |
| <code>-x</code> | Унарный минус |
| <code>+x</code> | Унарный плюс |
| <code>abs(x)</code> | Абсолютное значение |
| <code>divmod(x, y)</code> | Возвращает (x // y, x % y) |
| <code>pow(x, y [, modulo])</code> | Возвращает (x ** y) % modulo |
| <code>round(x, [n])</code> | Округляет до ближайшего кратного 10 |

Обычно они имеют числовую интерпретацию. Но есть примечательные частные случаи. Например, оператор `+` используется и для конкатенации последовательностей, оператор `*` дублирует последовательности, оператор `-` используется для вычитания множеств, а оператор `%` форматирует строки:

```
[1,2,3] + [4,5] # [1,2,3,4,5]
[1,2,3] * 4      # [1,2,3,1,2,3,1,2,3,1,2,3]
'%s has %d messages' % ('Dave', 37)
```

Проверка операторов — динамический процесс. Операции со смешанными типами данных часто работают, если их смысл понятен интуитивно. Например, можно складывать целые и дробные числа:

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = 5
```



```
>>> a + b
Fraction(17, 3)
>>>
```

Но идеальное решение находится не всегда. Например, оно не работает с `Decimal`:

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Decimal('5')
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Fraction' and 'decimal.
Decimal'
>>>
```

И все же для большинства комбинаций чисел Python следует стандартной числовой иерархии логических значений, целых, дробных, комплексных чисел и чисел с плавающей точкой. Операции смешанного типа просто работают — вам не придется для этого ничего делать.

2.4. ПРИСВАИВАНИЕ НА МЕСТЕ

Python поддерживает операции присваивания «на месте», или «комбинированные» операции присваивания. Они перечислены в табл. 2.2.

Такие формы записи не считаются выражениями. Это синтаксические удобства для обновления значений на месте. Пример:

```
a = 3
a = a + 1      # a = 4
a += 1         # a = 5
```

Изменяемые объекты могут использовать эти операторы для модификации данных на месте как средство оптимизации:

```
>>> a = [1, 2, 3]
>>> b = a      # Создает новую ссылку на a
>>> a += [4, 5] # Обновление на месте (без создания нового списка)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>>
```

Таблица 2.2. Комбинированные операции присваивания

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|----------------------------|-------------------------------|
| <code>x += y</code> | <code>x = x + y</code> |
| <code>x -= y</code> | <code>x = x - y</code> |
| <code>x *= y</code> | <code>x = x * y</code> |
| <code>x /= y</code> | <code>x = x / y</code> |
| <code>x //= y</code> | <code>x = x // y</code> |
| <code>x **= y</code> | <code>x = x ** y</code> |
| <code>x %= y</code> | <code>x = x % y</code> |
| <code>x @= y</code> | <code>x = x @ y</code> |
| <code>x &= y</code> | <code>x = x & y</code> |
| <code>x = y</code> | <code>x = x y</code> |
| <code>x ^= y</code> | <code>x = x ^ y</code> |
| <code>x >>= y</code> | <code>x = x >> y</code> |
| <code>x <<= y</code> | <code>x = x << y</code> |

В этом примере `a` и `b` — ссылки на один и тот же список. При выполнении операции `a += [4, 5]` объект списка обновляется на месте без создания нового списка. Это обновление отразится и на `b`. Часто это выглядит довольно странно.

2.5. СРАВНЕНИЕ ОБЪЕКТОВ

Оператор проверки равенства (`x == y`) проверяет, равны ли значения `x` и `y`. В случае списков и кортежей они должны иметь одинаковый размер и содержать одинаковые элементы, следующие в одинаковом порядке. Для словарей `True` возвращается, только если `x` и `y` содержат одинаковые множества ключей, а все объекты с одинаковыми ключами имеют равные значения. Два множества считаются равными, если содержат одинаковые элементы.

Проверка равенства между объектами несовместимых типов (например, файл и число с плавающей точкой) не вызывает ошибку, а возвращает `False`. Но иногда сравнение объектов разных типов выдает `True`. Например, это происходит при сравнении целого числа с числом с плавающей точкой такой же величины:

```
>>> 2 == 2.0
True
>>>
```

Операторы тождественности (`x is y` и `x is not y`) проверяют два значения, чтобы определить, ссылаются ли они на один объект в памяти (например, `id(x) == id(y)`). Возможны ситуации, когда `x == y`, но проверка `x is not y` дает положительный результат:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> a == b
True
>>>
```

На практике сравнивать объекты оператором `is` не стоит. Используйте оператор `==` для всех сравнений, если только у вас нет веских причин для обратного.

2.6. ОПЕРАТОРЫ ПОРЯДКОВОГО СРАВНЕНИЯ

Операторы порядкового сравнения из табл. 2.3 имеют стандартную математическую интерпретацию для чисел. Они возвращают логическое значение.

Таблица 2.3. Операторы сравнения

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|------------------------|------------------|
| <code>x < y</code> | Меньше |
| <code>x > y</code> | Больше |
| <code>x >= y</code> | Больше или равно |
| <code>x <= y</code> | Меньше или равно |

Для множеств `x < y` проверяет, является ли `x` строгим подмножеством `y` (то есть содержит меньше элементов, но не равно `y`).

При сравнении двух последовательностей сравниваются первые элементы каждой. Их неравенство определяет результат. Если же элементы равны, сравнение переходит ко второму элементу каждой последовательности. Процесс продолжается, пока не найдутся два разных элемента или элементы в одной из последовательностей не закончатся. Если сравнение дойдет до конца обеих последовательностей, они считаются равными. Если `a` является подпоследовательностью `b`, то `a < b`.

Строки и байты сравниваются по лексикографическому критерию. Каждому символу присваивается уникальный числовой индекс, определяемый

кодировкой (например ASCII или «Юникод»). Считается, что один символ меньше другого, если меньше его индекс.

Не все типы поддерживают порядковые сравнения. Например, попытка использовать < в словарях не определена и приводит к ошибке `TypeError`. То же и с применением порядковых сравнений к несовместимым типам (например, строке и числу).

2.7. ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ И КВАЗИИСТИННОСТЬ

Операторы `and`, `or` и `not` используются для построения сложных логических выражений. Поведение этих операторов описано в табл. 2.4.

Таблица 2.4. Логические операторы

| ОПЕРАТОР | ОПИСАНИЕ |
|----------------------|---|
| <code>x or y</code> | Если значение <code>x</code> ложно, возвращается <code>y</code> , в противном случае возвращается <code>x</code> |
| <code>x and y</code> | Если значение <code>x</code> ложно, возвращается <code>x</code> , в противном случае возвращается <code>y</code> |
| <code>not x</code> | Если значение <code>x</code> ложно, возвращается <code>True</code> , в противном случае возвращается <code>False</code> |

Когда выражение используется для определения истинного или ложного значения, любое ненулевое число, непустая строка, список, кортеж или словарь интерпретируется как истинное. `False`, нуль, `None` и пустые списки, кортежи и словари интерпретируются как ложное значение.

Логические (булевские) выражения вычисляются слева направо, а правый операнд используется, только если нужен для определения итогового значения. Например, в результате вычисления `a and b` вы получите `b`, только если `a` истинно. Здесь срабатывает так называемое ускоренное вычисление, которое может пригодиться для упрощения кода с проверкой и последующей операцией. Пример:

```
if y != 0:
    result = x / y
else:
    result = 0

# Альтернатива
result = y and x / y
```

Во второй версии деление x / y выполняется только при ненулевом значении y .

Опора на неявную «правдивость» объектов может привести к появлению трудно обнаруживаемых ошибок. Рассмотрим следующую функцию:

```
def f(x, items=None):
    if not items:
        items = []
    items.append(x)
    return items
```

Функция получает необязательный аргумент, при отсутствии которого создается и возвращается новый список:

```
>>> foo(4)
[4]
>>>
```

Но если функции в аргументе передается существующий пустой список, она начинает вести себя странно:

```
>>> a = []
>>> foo(3, a)
[3]
>>> a          # Обратите внимание: a НЕ обновляется
[]
>>>
```

Это ошибка проверки истинности. Пустые списки определяются как `False`, поэтому код создает новый список, а не использует переданный в аргументе (`a`). Чтобы избавиться от этой проблемы, необходимо точнее выполнить сравнение с `None`:

```
def f(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

Старайтесь как можно точнее формулировать условные проверки.

2.8. УСЛОВНЫЕ ВЫРАЖЕНИЯ

В программировании часто встречается условное присваивание значения в зависимости от результата выражения:

```

if a <= b:
    minvalue = a
else:
    minvalue = b

```

Этот код можно сократить с помощью условного выражения:

```
minvalue = a if a <= b else b
```

Здесь сначала вычисляется условие в середине. Если результат равен `True`, то вычисляется выражение слева, а если нет — справа. Секция `else` обязательна.

2.9. ОПЕРАЦИИ С ИТЕРИРУЕМЫМИ ОБЪЕКТАМИ

Перебор — важный механизм Python, поддерживаемый всеми его контейнерами (списками, кортежами, словарями и т. д.), файлами и функциями-генераторами. Операции в табл. 2.5 могут применяться к любому итерируемому объекту `s`.

Таблица 2.5. Операции с итерируемыми объектами

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|---|
| <code>for vars in s:</code> | Перебор |
| <code>v1, v2, ... = s</code> | Распаковка в переменные |
| <code>x in s, x not in s</code> | Проверка принадлежности |
| <code>[a, *s, b], (a, *s, b), {a, *s, b}</code> | Расширение в литерале списка, кортежа или множества |

Важнейшая операция с итерируемым объектом — цикл `for`, перебирающий значения одно за одним. Все остальные строятся на этой основе.

Оператор `x in s` проверяет присутствие `x` среди элементов, производимых итерируемым объектом `s`, и возвращает `True` или `False`. Оператор `x not in s` не равен `not (x in s)`. Для строк операторы `in` и `not in` получают подстроки. Например `'hello' in 'hello world'` возвращает `True`. Учтите, что оператор `in` не поддерживает подстановочные символы или какие-либо разновидности поиска по шаблону.

У любого итерируемого объекта значения можно распаковать в набор ячеек памяти:

```

items = [ 3, 4, 5 ]
x, y, z = items      # x = 3, y = 4, z = 5

```

```
letters = "abc"
x, y, z = letters    # x = 'a', y = 'b', z = 'c'
```

Ячейки памяти слева не обязательно должны быть простыми именами переменных. Допустима любая ячейка памяти, которая может располагаться слева от знака равенства. Код можно записать следующим образом:

```
items = [3, 4, 5]
d = { }
d['x'], d['y'], d['z'] = items
```

При распаковке значений в ячейки памяти их число в левой части должно точно соответствовать количеству элементов в итерируемом объекте в правой. Для вложенных структур данных сопоставляйте ячейки памяти и данные, следуя одному и тому же структурному шаблону.

Рассмотрим пример распаковки двух вложенных трехэлементных кортежей:

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, day, year), (hour, minute, am_pm) = datetime
```

Иногда переменная `_` используется для выражения игнорируемых значений при распаковке. Например, если вас интересует только день и час, можно добавить следующую запись:

```
(_, day, _), (hour, _, _) = datetime
```

Если количество распаковываемых элементов неизвестно заранее, используйте расширенную форму распаковки с переменной, имеющей префикс `*`:

```
items = [1, 2, 3, 4, 5]
a, b, *extra = items    # a = 1, b = 2, extra = [3,4,5]
*extra, a, b             # extra = [1,2,3], a = 4, b = 5
a, *extra, b             # a = 1, extra = [2,3,4], b = 5
```

В этом примере `*extra` содержит все остальные элементы. Это всегда список. При распаковке одного итерируемого объекта может использоваться только одна переменная с префиксом `*`. Но при распаковке более сложных структур данных, в которых задействованы разные итерируемые объекты, можно использовать несколько переменных с префиксом `*`. Например:

```
datetime = ((5, 19, 2008), (10, 30, "am"))

(month, *_), (hour, *_) = datetime
```

Любой итерируемый объект может расширяться в записи литералов списков, кортежей и множеств. При этом тоже используется символ * (звездочка):

```
items = [1, 2, 3]
a = [10, *items, 11]      # a = [10, 1, 2, 3, 11] (список)
b = (*items, 10, *items)  # b = [1, 2, 3, 10, 1, 2, 3] (кортеж)
c = {10, 11, *items}      # c = {1, 2, 3, 10, 11} (множество)
```

Здесь содержимое `items` просто вставляется в создаваемый список, кортеж или набор, как если бы вы ввели его в этом месте в ячейке памяти. При определении литерала можно использовать сколько угодно расширений *. Но многие итерируемые объекты (файлы или генераторы) поддерживают только одноразовый перебор. При использовании *-расширения содержимое будет потреблено, и итерируемый объект не будет производить новых значений при последующих итерациях.

Многие встроенные функции получают на входе любой итерируемый объект. Некоторые из этих операций перечислены в табл. 2.6.

Таблица 2.6. Функции, получающие итерируемые объекты

| ФУНКЦИЯ | ОПИСАНИЕ |
|---------------------------------|---|
| <code>list(s)</code> | Создает список на базе <code>s</code> |
| <code>tuple(s)</code> | Создает кортеж на базе <code>s</code> |
| <code>set(s)</code> | Создает множество на базе <code>s</code> |
| <code>min(s [,key])</code> | Минимальный элемент в <code>s</code> |
| <code>max(s [,key])</code> | Максимальный элемент в <code>s</code> |
| <code>any(s)</code> | Возвращает <code>True</code> , если хотя бы один элемент в <code>s</code> истинен |
| <code>all(s)</code> | Возвращает <code>True</code> , если все элементы в <code>s</code> истинны |
| <code>sum(s [, initial])</code> | Сумма элементов с необязательным начальным значением |
| <code>sorted(s [, key])</code> | Создает отсортированный список |

Это относится и ко многим библиотечным функциям, например функциям модуля `statistics`.

2.10. ОПЕРАЦИИ С ПОСЛЕДОВАТЕЛЬНОСТЯМИ

Последовательность — это итерируемый контейнер некоторого размера, поддерживающий обращение к элементам по целочисленным индексам,

начиная с 0. Примеры последовательностей — строки, списки и кортежи. Помимо всех операций, подразумевающих перебор, к последовательностям могут применяться операторы из табл. 2.7.

Таблица 2.7. Операции с последовательностями

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|--|
| <code>s + r</code> | Конкатенация |
| <code>s * n</code> , <code>n * s</code> | Создает <code>n</code> копий <code>s</code> , где <code>n</code> – целое число |
| <code>s[i]</code> | Индексирование |
| <code>s[i:j]</code> | Сегментация |
| <code>s[i:j:stride]</code> | Расширенная сегментация |
| <code>len(s)</code> | Длина |

Оператор `+` производит конкатенацию двух однотипных последовательностей:

```
>>> a = [3, 4, 5]
>>> b = [6, 7]
>>> a + b
[3, 4, 5, 6, 7]
>>>
```

Оператор `s * n` создает `n` копий последовательности. Учтите, что создаются поверхностные копии, дублирующие элементы только по ссылке. Рассмотрим следующий фрагмент:

```
>>> a = [3, 4, 5]
>>> b = [a]
>>> c = 4 * b
>>> c
[[3, 4, 5], [3, 4, 5], [3, 4, 5], [3, 4, 5]]
>>> a[0] = -7
>>> c
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
>>>
```

Обратите внимание на то, как изменение `a` приводит к изменению каждого элемента списка `c`. Здесь ссылка на список `a` включается в список `b`. При репликации `b` создаются четыре дополнительные ссылки на `a`. Изменение `a` распространяется на все остальные копии `a`. Такое поведение умножения последовательностей часто не входит в намерения программиста. Одно из обходных решений проблемы — ручное построение реплицированной последовательности с дублированием содержимого `a`:

```
a = [ 3, 4, 5 ]
c = [list(a) for _ in range(4)] # list() создает копию списка
```

Оператор индексирования `s[n]` возвращает n -й объект из последовательности, `s[0]` — первый объект. Отрицательные индексы могут использоваться для получения символов от конца последовательности. Например, `s[-1]` возвращает последний элемент. В остальных случаях попытки обращения к элементам за пределами диапазона приводят к исключению `IndexError`.

Оператор сегментации `s[i:j]` извлекает из `s` подпоследовательность из элементов с индексом k , где $i \leq k < j$. i и j должны быть целыми числами. Если начальный или конечный индекс опущен, предполагается начало или конец последовательности соответственно. Отрицательные индексы допустимы и определяются как заданные относительно конца последовательности.

Оператор сегментации может получать необязательное приращение `stride`, `s[i:j:stride]`, заставляющее сегмент пропускать элементы. Но его поведение при этом несколько усложняется. Если приращение `stride` задано, i — начальный индекс, j — конечный, то произведенная подпоследовательность содержит элементы `s[i]`, `s[i+stride]`, `s[i+2*stride]` и т. д., пока не будет достигнут индекс j (не включая последний). Приращение может быть отрицательным. Если начальный индекс i не задан, используется начало последовательности при положительном приращении или ее конец при отрицательном. Если конечный индекс j не задан, используется конец последовательности, если приращение положительно, или конец последовательности, если приращение отрицательно. Несколько примеров:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

a[2:5]    # [2, 3, 4]
a[:3]     # [0, 1, 2]
a[-3:]    # [7, 8, 9]
a[::2]    # [0, 2, 4, 6, 8 ]
a[::-2]    # [9, 7, 5, 3, 1 ]
a[0:5:2]   # [0, 2, 4]
a[5:0:-2]  # [5, 3, 1]
a[:5:1]    # [0, 1, 2, 3, 4]
a[:5:-1]   # [9, 8, 7, 6]
a[5::1]    # [5, 6, 7, 8, 9]
a[5::-1]   # [5, 4, 3, 2, 1, 0]
a[5:0:-1]  # [5, 4, 3, 2, 1]
```

Слишком сложные сегменты могут привести к появлению труднопони-
маемого кода. Поэтому при их использовании стоит руководствоваться

здравым смыслом. Сегментам можно присвоить имена при помощи функции `slice()`:

```
firstfive = slice(0, 5)
s = 'hello world'
print(s[firstfive])    # Выводит 'hello'
```

2.11. ОПЕРАЦИИ С ИЗМЕНЯЕМЫМИ ПОСЛЕДОВАТЕЛЬНОСТЯМИ

Строки и кортежи не могут изменяться после создания. Содержимое списка или другой изменяемой последовательности можно изменить на месте при помощи операторов из табл. 2.8.

Таблица 2.8. Операции с изменяемыми последовательностями

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--------------------------------|-------------------------------------|
| <code>s[i] = x</code> | Присваивание по индексу |
| <code>s[i:j] = r</code> | Сегментное присваивание |
| <code>s[i:j:stride] = r</code> | Расширенное сегментное присваивание |
| <code>del s[i]</code> | Удаляет элемент |
| <code>del s[i:j]</code> | Удаляет сегмент |
| <code>del s[i:j:stride]</code> | Удаляет расширенный сегмент |

Оператор `s[i] = x` изменяет элемент `i` последовательности и сохраняет в нем ссылку на объект `x`, увеличивая счетчик ссылок `x`. Отрицательные индексы отсчитываются от конца списка, а попытка присвоить значение по индексу за пределами диапазона приводит к исключению `IndexError`. Оператор сегментного присваивания `s[i:j] = r` заменяет элементы `k`, где $i \leq k < j$, элементами из последовательности `r`. Индексы имеют то же значение, что и при сегментации. При необходимости последовательность `s` может расширяться или сокращаться, чтобы вмещать все элементы `r`:

```
a = [1, 2, 3, 4, 5]
a[1] = 6                # a = [1, 6, 3, 4, 5]
a[2:4] = [10, 11]       # a = [1, 6, 10, 11, 5]
a[3:4] = [-1, -2, -3]    # a = [1, 6, 10, -1, -2, -3, 5]
a[2:] = [0]              # a = [1, 6, 0]
```

При сегментном присваивании может передаваться необязательный аргумент `stride`. Но поведение в этом случае ограничивается: число элементов

в аргументе справа должно точно соответствовать числу элементов в замещаемом сегменте. Пример:

```
a = [1, 2, 3, 4, 5]
a[1::2] = [10, 11]      # a = [1, 10, 3, 11, 5]
a[1::2] = [30, 40, 50] # ValueError. В сегменте слева только
                        # два элемента
```

Оператор `del s[i]` удаляет элемент `i` из последовательности и уменьшает счетчик его ссылок. `del s[i:j]` удаляет все элементы в сегменте. Еще можно передать приращение, как в `del s[i:j:stride]`.

Описанная семантика относится к встроенному типу списка. Операции, связанные с сегментацией последовательностей, открывают широкие возможности для настройки в сторонних пакетах. При сегментации других объектов, кроме списков, могут действовать другие правила присваивания, удаления и совместного использования объектов. Например, в популярном пакете `numpy` семантика сегментации отличается от семантики списков Python.

2.12. ОПЕРАЦИИ С МНОЖЕСТВАМИ

Множество — это неупорядоченная коллекция уникальных значений. Операции с множествами перечислены в табл. 2.9.

Таблица 2.9. Операции с множествами

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|---|
| <code>s t</code> | Объединение <code>s</code> и <code>t</code> |
| <code>s & t</code> | Пересечение <code>s</code> и <code>t</code> |
| <code>s - t</code> | Разность множеств (элементы, присутствующие в <code>s</code> , но не в <code>t</code>) |
| <code>s ^ t</code> | Симметричная разность (элементы, не присутствующие одновременно в <code>s</code> и <code>t</code>) |
| <code>len(s)</code> | Количество элементов в множестве |
| <code>item in s</code> , <code>item not in s</code> | Проверка принадлежности |
| <code>s.add(item)</code> | Добавляет элемент в множество <code>s</code> |
| <code>s.remove(item)</code> | Удаляет элемент из <code>s</code> , если он существует (в противном случае происходит ошибка) |
| <code>s.discard(item)</code> | Удаляет элемент из <code>s</code> , если он существует |

Несколько примеров:

```
>>> a = {'a', 'b', 'c' }
>>> b = {'c', 'd'}
>>> a | b
{'a', 'b', 'c', 'd'}
>>> a & b
{'c' }
>>> a - b
{'a', 'b'}
>>> b - a
{'d' }
>>> a ^ b
{'a', 'b', 'd' }
>>>
```

Операции с множествами работают и с объектами представления ключей и элементов словарей. Например, узнать общие ключи в обоих словарях можно так:

```
>>> a = { 'x': 1, 'y': 2, 'z': 3 }
>>> b = { 'z': 3, 'w': 4, 'q': 5 }
>>> a.keys() & b.keys()
{ 'z' }
>>>
```

2.13. ОПЕРАЦИИ С ОТОБРАЖЕНИЯМИ

Отображение (*mapping*) — это связь между ключами и значениями. Например, встроенный тип `dict`. Операции в табл. 2.10 могут применяться к отображениям.

Таблица 2.10. Операции с отображениями

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|-------------------------|------------------------------------|
| <code>x = m[k]</code> | Индексирование по ключу |
| <code>m[k] = x</code> | Присваивание по ключу |
| <code>del m[k]</code> | Удаляет элемент по ключу |
| <code>k in m</code> | Проверка принадлежности |
| <code>len(m)</code> | Количество элементов в отображении |
| <code>m.keys()</code> | Возвращает ключи |
| <code>m.values()</code> | Возвращает значения |
| <code>m.items()</code> | Возвращает пары (ключ, значение) |

Значениями ключей могут быть любые неизменяемые объекты — строки, числа и кортежи. При использовании кортежа в качестве ключа можно опустить круглые скобки и записать значения, разделенные запятыми:

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

Здесь значения ключей представляют кортеж, а эти присваивания эквивалентны следующим:

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

Использование кортежа в качестве ключа — стандартный способ создания составных ключей в отображении (mapping). Например, ключ может состоять из имени и фамилии.

2.14. ВКЛЮЧЕНИЯ СПИСКОВ, МНОЖЕСТВ И СЛОВАРЕЙ

Одна из самых распространенных операций с данными — преобразование набора данных в другую структуру. В следующем примере мы берем все элементы списка, применяем к ним какую-то операцию и создаем новый список:

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    nums.append(n * n)
```

Из-за частоты выполнения таких операций для них был создан специальный оператор — списковое включение. Более компактная версия этого кода выглядит так:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

Можно также применить к операции фильтр:

```
squares = [n * n for n in nums if n > 2] # [9, 16, 25]
```

Обобщенный синтаксис спискового включения выглядит так:

```
[выражение for элемент1 in итерируемый1 if условие1
    for элемент2 in итерируемый2 if условие2
...
    for элементN in итерируемыйN if условиеN ]
```

Этот синтаксис аналогичен следующему коду:

```
result = []
for элемент1 in итерируемый1:
    if условие1:
        for элемент2 in итерируемый2:
            if условие2:
                ...
                for элементN in итерируемыйN:
                    if условиеN:
                        result.append(выражение)
```

Понимание списков — очень полезный способ обработки данных списков в разных формах. Несколько практических примеров:

```
# Данные (список словарей)
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1 },
    {'name': 'MSFT', 'shares': 50, 'price': 45.67 },
    {'name': 'HPE', 'shares': 75, 'price': 34.51 },
    {'name': 'CAT', 'shares': 60, 'price': 67.89 },
    {'name': 'IBM', 'shares': 200, 'price': 95.25 }
]

# Получить все имена ['IBM', 'MSFT', 'HPE', 'CAT', 'IBM' ]
names = [s['name'] for s in portfolio]

# Найти все записи с более чем 100 акциями ['IBM']
more100 = [s['name'] for s in portfolio if s['shares'] > 100 ]

# Найти сумму shares*price
cost = sum([s['shares']*s['price'] for s in portfolio])

# Получить кортежи (name, shares)
name_shares = [ (s['name'], s['shares']) for s in portfolio ]
```

Все переменные, используемые в списковом включении, приватны по отношению к включению. Вам не нужно беспокоиться о том, не заместят ли они другие переменные с такими же именами. Пример:

```
>>> x = 42
>>> squares = [x*x for x in [1,2,3]]
```

```
>>> squares
[1, 4, 9]
>>> x
42
>>>
```

Вместо списка можно создать множество. Для этого нужно заменить квадратные скобки фигурными. Созданный так генератор множеств предоставляет множество с различающимися значениями. Пример:

```
# Включение множества
names = { s['name'] for s in portfolio }
# names = { 'IBM', 'MSFT', 'HPE', 'CAT' }
```

Если же задать пары «ключ — значение», вы создадите словарь. Это называется словарным включением. Пример:

```
prices = { s['name']:s['price'] for s in portfolio }
# prices = { 'IBM': 95.25, 'MSFT': 45.67, 'HPE': 34.51, 'CAT': 67.89 }
```

При создании множеств и словарей помните, что более поздние элементы могут заменять более ранние. Например, в словаре `prices` вы получите последнюю цену для акций IBM, а первая будет потеряна.

Во включения нельзя встроить обработку исключений. Если это создает проблемы, стоит упаковать исключения в функцию:

```
def toint(x):
    try:
        return int(x)
    except ValueError:
        return None

values = [ '1', '2', '-4', 'n/a', '-3', '5' ]
data1 = [ toint(x) for x in values ]
# data1 = [1, 2, -4, None, -3, 5]

data2 = [ toint(x) for x in values if toint(x) is not None ]
# data2 = [1, 2, -4, -3, 5]
```

Для предотвращения двойного вычисления `toint(x)` в последнем примере используйте оператор `:=`:

```
data3 = [ v for x in values if (v:=toint(x)) is not None ]
# data3 = [1, 2, -4, -3, 5]
data4 = [ v for x in values if (v:=toint(x)) is not None and v >= 0 ]
# data4 = [1, 2, 5]
```


2.15. ВЫРАЖЕНИЯ-ГЕНЕРАТОРЫ

Выражение-генератор — это объект, выполняющий то же вычисление, что и списковое включение, но выдающий результат в итеративной форме. Синтаксис схож с синтаксисом спискового включения, но вместо квадратных скобок используются круглые. Пример:

```
nums = [1,2,3,4]
squares = (x*x for x in nums)
```

В отличие от спискового включения, выражение-генератор не создает список и не вычисляет выражение в круглых скобках немедленно. Вместо этого оно создает объект-генератор, производящий значения по требованию. Заглянув в результат примера выше, вы увидите следующее:

```
>>> squares
<generator object at 0x590a8>
>>> next(squares)
1
>>> next(squares)
4
...
>>> for n in squares:
...     print(n)
9
16
>>>
```

Выражение-генератор можно использовать только один раз. Вторая попытка не даст никакого результата:

```
>>> for n in squares:
...     print(n)
...
>>>
```

Между списковыми включениями и выражениями-генераторами есть важные, но неочевидные различия. Со списковым включением Python создает реальный список с итоговыми данными. С выражением-генератором Python создает генератор, способный только производить данные по требованию. В некоторых приложениях это может сильно улучшить производительность и эффективность использования памяти. Пример:

| | |
|--------------------------------|-------------------------------|
| # Чтение файла | # Открыть файл |
| f = open('data.txt') | # Прочитать строки, удалить |
| lines = (t.strip() for t in f) | # начальные/конечные пропуски |

```

comments = (t for t in lines if t[0] == '#') # Все комментарии
for c in comments:
    print(c)

```

В этом примере выражение-генератор, извлекающее строки и удаляющее пропуски, обходится без чтения и хранения всего файла в памяти. То же касается и выражения, извлекающего комментарии. Вместо этого строки файла читаются по одной, когда программа начинает перебор в следующем цикле `for`. В ходе этого перебора строки файла создаются по запросу и соответственно фильтруются. Так ни в какой точке этого процесса файл не будет находиться в памяти целиком. А значит, такой способ извлечения комментариев из гигабайтных файлов с исходным кодом Python очень эффективен.

В отличие от списковых включений, выражение-генератор не создает объект, работающий как последовательность. Он не может индексироваться, и никакие обычные операции списков (например, `append()`) с ним не работают. Но элементы, производимые выражением-генератором, можно преобразовать в список вызовом `list()`:

```
clist = list(comments)
```

При передаче функции одного аргумента можно удалить один набор скобок. Например, следующие команды равнозначны:

```

sum((x*x for x in values))
sum(x*x for x in values)   # Лишние круглые скобки удалены

```

В обоих случаях будет создан генератор `(x*x for x in values)`, который передается функции `sum()`.

2.16. ОПЕРАТОР АТТРИБУТА (.)

Оператор «точка» (.) используется для обращения к атрибутам объектов. Пример:

```

foo.x = 3
print(foo.y)
a = foo.bar(3,4,5)

```

В одном выражении может быть несколько этих операторов (например, `foo.y.a.b`). Оператор может применяться и к промежуточным результатам функций, как в команде `a = foo.bar(3,4,5).spam`. Но со стилистической

точки зрения длинные цепочки обращений к атрибутам встречаются в программах редко.

2.17. ОПЕРАТОР ВЫЗОВА ФУНКЦИИ ()

Оператор `f(...)` используется для вызова функции `f`. Каждый аргумент функции является выражением. До вызова функции все аргументы-выражения полностью вычисляются слева направо. Это называют аппликативным порядком вычисления. Дополнительную информацию о функциях см. в главе 5.

2.18. ПОРЯДОК ВЫЧИСЛЕНИЯ

В табл. 2.11 перечислены правила приоритета (порядок выполнения операций) для операторов Python. Все они, кроме оператора возведения в степень (`**`), вычисляются слева направо. В таблице они перечислены по убыванию приоритета. Иначе говоря, операторы, указанные ближе к началу таблицы, вычисляются до тех, что указаны позднее. У операторов из одного подраздела (например, `x * y`, `x / y`, `x // y`, `x @ y` и `x % y`) одинаковый приоритет.

Порядок вычисления из табл. 2.11 не зависит от типов `x` и `y`. И хотя объекты, определяемые пользователем, могут переопределять отдельные операторы, невозможно изменить используемый порядок вычисления, приоритеты и правила ассоциативности.

Таблица 2.11. Порядок вычисления (от наибольшего приоритета к наименьшему)

| ОПЕРАТОР | НАЗВАНИЕ |
|---|---|
| <code>(...), [...], {...}</code> | Создание кортежей, списков и словарей |
| <code>s[i], s[i:j]</code> | Индексирование и сегментация |
| <code>s.attr</code> | Обращение к атрибуту |
| <code>f(...)</code> | Вызов функции |
| <code>+x, -x, ~x</code> | Унарные операторы |
| <code>x ** y</code> | Возведение в степень (правосторонняя ассоциативность) |
| <code>x * y, x / y, x // y, x % y, x @ y</code> | Умножение, деление, целочисленное деление, вычисление остатка, умножение матриц |

Таблица 2.11 (окончание)

| ОПЕРАТОР | НАЗВАНИЕ |
|---|---|
| <code>x + y, x - y</code> | Сложение, вычитание |
| <code>x << y, x >> y</code> | Битовый сдвиг |
| <code>x & y</code> | Битовая операция И |
| <code>x ^ y</code> | Битовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ |
| <code>x y</code> | Битовая операция ИЛИ |
| <code>x < y, x <= y, x > y, x >= y, x == y, x != y, x is y, x is not y, x in y, x not in y</code> | Сравнение, тождественность и проверка принадлежности к последовательности |
| <code>not x</code> | Логическое отрицание |
| <code>x and y</code> | Логическая операция И |
| <code>x or y</code> | Логическая операция ИЛИ |
| <code>lambda args: expr</code> | Анонимная функция |
| <code>expr if expr else expr</code> | Условное выражение |
| <code>name := expr</code> | Выражение с присваиванием |

Типичная ошибка с правилами приоритета: операторы битовой операции И (&) и битовой операции ИЛИ (|) используются вместо логической операции И (and) и логической операции ИЛИ (or). Пример:

```
>>> a = 10
>>> a <= 10 and 1 < a
True
>>> a <= 10 & 1 < a
False
>>>
```

Последнее выражение вычисляется как `a <= (10 & 1) < a` or `a <= 0 < a`. Проблему можно решить добавлением круглых скобок:

```
>>> (a <= 10) & (1 < a)
True
>>>
```

На первый взгляд кажется, что это необычный случай, но он нередко встречается в таких объектно-ориентированных пакетах, как `numpy` и `pandas`. Логические операторы `and` и `or` невозможно настроить так, чтобы вместо них

использовались битовые операторы, несмотря на их более высокий уровень приоритета и иной способ вычисления в логических отношениях.

2.19. НАПОСЛЕДОК: ТАЙНАЯ ЖИЗНЬ ДАННЫХ

Одна из наиболее частых областей применения Python — приложения, ориентированные на обработку и анализ данных. В этой области Python — своего рода язык предметной области для размышлений над задачей. Встроенные операторы и выражения — основа языка, на которой строится все остальное. Когда у вас сформируются интуитивные представления о встроенных объектах и операциях Python, вы увидите, что они применимы повсюду.

Допустим, вы работаете с базой данных и хотите перебрать записи, возвращенные запросом. Скорее всего, вы используете для этого команду `for`. Или представьте, что вы работаете с числовыми массивами и хотите выполнить поэлементные математические вычисления с одним из них. Вы предположите, что для этого можно воспользоваться стандартными математическими операторами — и это будет верно. Или, скажем, вы используете библиотеку для получения данных по протоколу HTTP и хотите обратиться к содержанию его заголовков. Скорее всего, данные будут представлены в формате, напоминающем словарь.

Больше информации о внутренних протоколах Python и их настройке вы найдете в главе 4.

ГЛАВА 3

Структура программы и управление последовательностью выполнения

В этой главе мы подробнее рассмотрим структуру программы и управление последовательностью выполнения. В частности, мы обсудим такие темы, как условия, циклы, исключения и менеджеры контекста.

3.1. СТРУКТУРА ПРОГРАММЫ И ВЫПОЛНЕНИЕ

Программы Python — это последовательности команд. Все основные языковые средства, включая присваивание, выражения, определения функций, классы и импорт модулей, — это команды, которые по своему статусу не отличаются от других. Это значит, что любая команда может находиться почти в любом месте программы (хотя некоторые, например `return`, могут быть только внутри функций). Следующий код определяет две разные версии функции внутри условной команды:

```
if debug:
    def square(x):
        if not isinstance(x, float):
            raise TypeError('Expected a float')
        return x * x
else:
    def square(x):
        return x * x
```

При загрузке исходных файлов интерпретатор выполняет команды по порядку, пока не будут выполнены все. Эта модель применяется и к файлам, выполняемым как основная программа, и к файлам библиотек, загружаемым командой `import`.

3.2. УСЛОВНОЕ ВЫПОЛНЕНИЕ

Команды `if`, `else` и `elif` управляют условным выполнением кода. Общий формат условной команды выглядит так:

```
if выражение:
    команды
elif выражение:
    команды
elif выражение:
    команды
...
else:
    команды
```

Если никакие действия выполняться не должны, секции `else` и `elif` условной команды можно опустить. Если в какой-то секции никакие команды не выполняются, используйте команду `pass`:

```
if expression:
    pass # TODO: Реализовать!
else:
    команды
```

3.3. ЦИКЛЫ И ПЕРЕБОР

Циклы реализуются командами `for` и `while`:

```
while выражение:
    команды

for i in s:
    команды
```

`while` выполняет команды до того, как указанное выражение даст результат `false`. `for` перебирает все элементы `s`, пока их не останется. Команда `for` работает с любым объектом, поддерживающим перебор. К этой категории относятся не только встроенные типы последовательностей (списки, кортежи и строки), но и любые объекты, реализующие протокол итератора.

В команде `for i in s` переменная `i` называется итеративной. При каждой итерации цикла она получает новое значение из `s`. Масштабирование итеративной переменной не приватно для `for`. Если есть ранее определенная переменная с таким же именем, это значение будет заменено. Более того, итеративная переменная сохраняет последнее значение по завершении цикла.

Если элементы, произведенные при переборе, — итерируемые объекты идентичного размера, их значения можно распаковать в отдельные итеративные переменные следующей командой:

```
s = [ (1, 2, 3), (4, 5, 6) ]
```

```
for x, y, z in s:
    команды
```

Здесь объект `s` должен содержать или производить итерации, состоящие из трех элементов. При каждой итерации переменным `x`, `y` и `z` присваиваются элементы соответствующего итерируемого объекта. Этот синтаксис чаще всего используется, когда `s` является последовательностью кортежей. Но распаковка работает, когда элементы `s` относятся к любой разновидности итерируемых объектов, включая списки, генераторы и строки.

Иногда при распаковке используются игнорируемые переменные `_`:

```
for x, _, z in s:
    команды
```

Здесь значение помещается в переменную `_`, но ее имя указывает, что она не представляет интереса и не используется в дальнейших вычислениях.

Если элементы, производимые итерируемым объектом, разных размеров, можно воспользоваться переменной с префиксом `*`, где сохраняются несколько значений. Пример:

```
s = [ (1, 2), (3, 4, 5), (6, 7, 8, 9) ]
for x, y, *extra in s:
    команды
    # x = 1, y = 2, extra = []
    # x = 3, y = 4, extra = [5]
    # x = 6, y = 7, extra = [8, 9]
    # ...
```

В этом примере важны как минимум два значения — `x` и `y`, а в переменную `*extra` помещаются все дополнительные, которые тоже могут быть в переборе. Эти значения всегда помещаются в список. В одной распаковке может участвовать только одна `*`-переменная, но она может находиться в произвольной позиции. Поэтому допустимы оба следующих варианта:


```

for *first, x, y in s:
    ...

for x, *middle, y in s:
    ...

```

При переборе иногда бывает полезно отслеживать не только значения данных, но и числовой индекс:

```

i = 0
for x in s:
    команды
    i += 1

```

Python предоставляет встроенную функцию `enumerate()`, которая может использоваться для упрощения этого кода:

```

for i, x in enumerate(s):
    команды

```

`enumerate(s)` создает итератор, производящий кортежи `(0, s[0])`, `(1, s[1])`, `(2, s[2])` и т. д. Другое начальное значение для отсчета может быть передано в ключевом аргументе `start` функции `enumerate()`:

```

for i, x in enumerate(s, start=100):
    команды

```

В этом случае будут производиться кортежи вида `(100, s[0])`, `(101, s[1])` и т. д.

При работе с циклами часто встречается проблема параллельного перебора двух и более итерируемых объектов. Вот пример цикла, где при каждой итерации получают элементы из разных последовательностей:

```

# s и t - две последовательности
i = 0
while i < len(s) and i < len(t):
    x = s[i]      # Получить элемент из s
    y = t[i]      # Получить элемент из t
    команды
    i += 1

```

Этот код можно упростить с помощью функции `zip()`:

```

# s и t - две последовательности
for x, y in zip(s, t):
    команды

```

`zip(s, t)` объединяет итерируемые `s` и `t` в итерируемый объект с кортежами `(s[0], t[0])`, `(s[1], t[1])`, `(s[2], t[2])` и т. д., останавливаясь на более коротком наборе из `s` и `t`, если они разной длины. Результат `zip()` — итератор, производящий результаты при переборе. Чтобы преобразовать результат в список, используйте `list(zip(s, t))`.

Для прерывания цикла используется команда `break`. Следующий код читает строки текста из файла, пока не обнаружит пустую строку:

```
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            break # Пустая строка, прервать чтение
        # Обработать усеченную строку
    ...
```

Для перехода к следующей итерации цикла (пропуская остаток тела цикла) используйте команду `continue`. Она удобна, когда обратная проверка и отступ на новый уровень приведут к слишком глубокой вложенности или излишнему усложнению кода. Следующий цикл пропускает все пустые строки в файле:

```
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            continue # Пропустить пустую строку
        # Обработать усеченную строку
    ...
```

Команды `break` и `continue` применяются только к внутреннему выполняемому циклу. Для выхода из структуры циклов с большой вложенностью используйте исключение. Python не поддерживает команды `goto`. К конструкции цикла можно присоединить и команду `else`, как в следующем примере:

```
# for-else
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            break
        # Обработать усеченную строку
    ...
else:
    raise RuntimeError('Missing section separator')
```

Секция `else` цикла выполняется только в случае его доведения до конца. Это происходит немедленно (если цикл вообще не выполняется) или после последней итерации. Если цикл преждевременно завершается командой `break`, то секция `else` пропускается.

Секция `else` у циклов используется в основном в коде, который перебирает данные, но при этом должен устанавливать или проверять некоторый флаг или условие при преждевременном завершении цикла. Например, без `else` предыдущий код можно было бы переписать с переменной-флагом так:

```
found_separator = False

with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            found_separator = True
            break
        # Обработать усеченную строку
    ...
if not found_separator:
    raise RuntimeError('Missing section separator')
```

3.4. ИСКЛЮЧЕНИЯ

Исключения сообщают о появлении ошибок и прерывают нормальную последовательность выполнения программы. Они выдаются командой `raise`. Обобщенный формат команды `raise` — `raise Exception([значение])`, где `Exception` — тип исключения, а `value` — необязательное значение с подробной информацией об исключении:

```
raise RuntimeError('Unrecoverable Error')
```

Для перехвата исключений используются команды `try` и `except`:

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError as e:
    команды
```

При появлении исключения интерпретатор перестает выполнять команды в блоке `try` и ищет секцию `except`, соответствующую типу возникшего исключения. Если такая секция будет найдена, управление передается первой команде в секции `except`. После выполнения секции `except` управление

продолжается с первого оператора, который появляется после всего блока `try-except`.

Команда `try` не обязана перехватывать все возможные исключения. Если подходящая секция `except` не найдена, исключение продолжает распространяться. Оно может быть перехвачено в другом блоке `try-except`, который обработает его в другом месте. Для соблюдения хорошего тона в программировании рекомендуют перехватывать только те исключения, после которых можно восстановить работоспособность программы. Если восстановление невозможно, часто лучше позволить исключению распространиться.

Когда исключение поднимается до верхнего уровня программы без перехвата, интерпретатор прерывает выполнение программы сообщением об ошибке.

Если команда `raise` используется сама по себе, то последнее сгенерированное исключение выдается снова. Этот прием работает только при обработке ранее выданного исключения. Пример:

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError:
    print("Well, that didn't work.")
    raise      # Заново выдает текущее исключение
```

Каждая секция `except` может использоваться с модификатором `as var`, задающим имя переменной, куда помещается экземпляр типа исключения при возникновении исключения. Обработчики могут проверить это значение, чтобы больше узнать о причине исключения. Для проверки типа исключения можно воспользоваться `isinstance()`.

У исключений несколько стандартных атрибутов, которые будут полезны в коде, выполняющем дополнительные действия в ответ на возникновение ошибки.

- `e.args` — кортеж аргументов, переданных при выдаче исключения. В большинстве случаев кортеж содержит только строку с описанием ошибки. Для исключений `OSError` значение — это кортеж из двух-трех элементов, содержащий целый номер ошибки, строку с сообщением о ней и необязательное имя файла.
- `e.__cause__` — предыдущее исключение, если оно было преднамеренно вызвано в ответ на обработку другого исключения. За дополнительной информацией обращайтесь к разделу, посвященному цепочкам исключений.

- `e.__context__` — предыдущее исключение, если оно было выдано при обработке другого исключения.
- `e.__traceback__` — объект трассировки стека, связанный с исключением.

Переменная для хранения значения исключения доступна только внутри соответствующего блока `except`. После того как управление выходит за пределы блока, она становится неопределенной:

```
try:
    int('N/A')          # Выдает ValueError
except ValueError as e:
    print('Failed:', e)

print(e)               # Ошибка -> NameError. Переменная 'e' не определена
```

Можно задать несколько блоков обработки исключений. Для этого используется несколько секций `except`:

```
try:
    что-то сделать
except TypeError as e:
    # Обработать ошибку TypeError
    ...
except ValueError as e:
    # Обработать ошибку ValueError
    ...
```

Одна секция-обработчик может перехватывать несколько типов исключений:

```
try:
    что-то сделать
except (TypeError, ValueError) as e:
    # Обработать ошибку TypeError или ValueError
    ...
```

Для игнорирования исключения используйте команду `pass`:

```
try:
    что-то сделать
except ValueError:
    pass                # Ничего не делать
```

Тихо игнорировать ошибки может быть опасно, ведь они становятся источником трудноуловимых сбоев. Даже если ошибка игнорируется, обычно

стоит зарегистрировать ее в журнале или другом месте, где ее можно будет потом проанализировать.

Чтобы перехватить все ошибки, кроме связанных с завершением программы, используйте `Exception`:

```
try:
    что-то сделать
except Exception as e:
    print(f'An error occurred : {e!r}')
```

При перехвате всех исключений очень внимательно сообщайте пользователю точную информацию об ошибках. Например, в прошлом фрагменте выводится сообщение об ошибке и связанное с ним значение исключения. Отсутствие информации о значении исключения сильно усложнит отладку кода, в котором происходят ошибки по каким-то непредвиденным причинам.

Команда `try` поддерживает секцию `else`, которая должна следовать за последней секцией `except`. Код `else` выполняется, если код блока `try` был выполнен без исключений:

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError as e:
    print(f'Unable to open foo : {e}')
    data = ''
else:
    data = file.read()
    file.close()
```

Команда `finally` определяет завершающее действие, которое должно быть выполнено независимо от того, что происходило в блоке `try-except`. Пример:

```
file = open('foo.txt', 'rt')
try:
    # Что-то сделать
    ...
finally:
    file.close()
    # Файл закрывается независимо от того, что происходило
```

Секция `finally` не используется для перехвата ошибок. Вместо этого в ней размещается код, который должен выполняться всегда, независимо от того, произошла ошибка или нет. Если исключение не выдается, то код секции `finally` выполняется сразу после кода блока `try`. Если же исключение было

выдано, сначала выполняется подходящий блок `try` (если он есть), а затем управление передается первой команде секции `finally`. Если после выполнения кода исключение все еще остается, оно выдается заново для перехвата другим обработчиком исключения.

3.4.1. Иерархия исключений

Одна из трудностей обработки исключений — множество исключений, которые теоретически могут возникнуть в вашей программе. Только встроенных более 60. Добавьте к ним остальные исключения стандартной библиотеки, и количество возможных вариантов увеличится до нескольких сотен. Более того, часто нельзя легко заранее определить, какие исключения могут возникнуть в той или иной части кода. Исключения не записываются в сигнатуре вызова функции, и компилятор не может гарантировать правильность обработки исключений в вашем коде. В результате обработка исключений иногда кажется хаотичной и непредсказуемой.

Полезно понимать, что исключения объединяются в иерархию, основанную на наследовании. Вместо того чтобы ориентироваться на конкретные ошибки, бывает проще сосредоточиться на более общих их категориях. Для примера можно взять разные ошибки, возникающие при получении значений из контейнера:

```
try:
    item = items[index]
except IndexError:      # Выдается, если items - последовательность
    ...
except KeyError:       # Выдается, если items - отображение
    ...
```

Чтобы не писать код обработки двух конкретных исключений, проще поступить так:

```
try:
    item = items[index]
except LookupError:
    ...
```

`LookupError` — класс, представляющий группу исключений на более высоком уровне. `IndexError` и `KeyError` наследуют от `LookupError`, так что секция `except` будет перехватывать оба исключения. Но категория `LookupError` не настолько широка, чтобы включать ошибки, не связанные с выборкой значений.

Самые распространенные категории встроенных исключений перечислены в табл. 3.1.

Таблица 3.1. Категории исключений

| КЛАСС ИСКЛЮЧЕНИЯ | ОПИСАНИЕ |
|------------------------------|---|
| <code>BaseException</code> | Корневой класс для всех исключений |
| <code>Exception</code> | Базовый класс для всех ошибок, связанных с выполнением программы |
| <code>ArithmeticError</code> | Базовый класс для ошибок, связанных с математическими вычислениями |
| <code>ImportError</code> | Базовый класс для ошибок, связанных с импортом |
| <code>LookupError</code> | Базовый класс для ошибок, связанных с выборкой из контейнера |
| <code>OSError</code> | Базовый класс для всех ошибок, связанных с системой. <code>OSError</code> и <code>EnvironmentError</code> являются синонимами |
| <code>ValueError</code> | Базовый класс для ошибок, связанных со значениями, включая «Юникод» |
| <code>UnicodeError</code> | Базовый класс для ошибок, связанных с кодировкой строк в «Юникоде» |

Класс `BaseException` редко используется при обработке исключений. Он представляет любые возможные исключения. К этой категории относятся специализированные исключения, влияющие на последовательность выполнения программы: `SystemExit`, `KeyboardInterrupt` и `StopIteration`. Перехватывать их обычно не нужно. Вместо этого все нормальные ошибки, относящиеся к программе, наследуют от `Exception`. Класс `ArithmeticError` базовый для всех ошибок, связанных с математическими вычислениями, таких как `ZeroDivisionError`, `FloatingPointError` и `OverflowError`. Класс `ImportError` базовый для всех ошибок, связанных с импортом. Класс `LookupError` базовый для всех ошибок, связанных с выборкой из контейнеров. Класс `OSError` базовый для всех ошибок, порожденных операционной системой и средой. Класс `OSError` охватывает широкий диапазон исключений, относящихся к файлам, сетевым подключениям, разрешениям, каналам, тайм-ауту и т. д. Исключение `ValueError` обычно выдается при передаче операции некорректного входного значения. `UnicodeError` — субкласс `ValueError`, объединяющий все ошибки, связанные с кодированием и декодированием «Юникода».

В табл. 3.2 вы увидите некоторые часто встречающиеся исключения, которые наследуются от `Exception`, но не входят в более крупные группы исключений.

Таблица 3.2. Другие встроенные исключения

| КЛАСС ИСКЛЮЧЕНИЯ | ОПИСАНИЕ |
|----------------------------------|---|
| <code>AssertionError</code> | Неудачная проверка <code>assert</code> |
| <code>AttributeError</code> | Неудача при поиске атрибута у объекта |
| <code>EOFError</code> | Конец файла |
| <code>MemoryError</code> | Ошибка нехватки памяти с возможностью восстановления |
| <code>NameError</code> | Имя не найдено в глобальном или локальном пространстве имен |
| <code>NotImplementedError</code> | Нереализованная возможность |
| <code>RuntimeError</code> | Обобщенная ошибка «Произошло что-то плохое» |
| <code>TypeError</code> | Операция применяется к объекту неправильного типа |
| <code>UnboundLocalError</code> | Локальная переменная используется до присваивания значения |

3.4.2. Исключения и последовательность выполнения

Обычно исключения резервируются для обработки ошибок. Но некоторые используются для изменения последовательности выполнения. Эти исключения, перечисленные в табл. 3.3, наследуются от `BaseException` напрямую.

Таблица 3.3. Исключения для управления последовательностью выполнения

| КЛАСС ИСКЛЮЧЕНИЯ | ОПИСАНИЕ |
|--------------------------------|--|
| <code>SystemExit</code> | Выдается для обозначения выхода из программы |
| <code>KeyboardInterrupt</code> | Выдается при прерывании программы сочетанием <code>Ctrl+C</code> |
| <code>StopIteration</code> | Выдается для обозначения завершения перебора |

Исключение `SystemExit` используется для намеренного завершения программ. В аргументе передается либо целочисленный код завершения, либо строковое сообщение. Если передается строка, она выводится в `sys.stderr`, после чего программа завершается с кодом 1. Типичный пример:

```
import sys
if len(sys.argv) != 2:
    raise SystemExit(f'Usage: {sys.argv[0]} filename')
filename = sys.argv[1]
```

Исключение `KeyboardInterrupt` выдается, когда программа получает сигнал `SIGINT` (обычно нажатием `Control+C` на терминале). Оно необычно тем, что происходит асинхронно: может появиться почти в любое время и в любой команде вашей программы. По умолчанию Python просто завершает программу, когда это происходит. Если вы хотите управлять доставкой `SIGINT`, используйте библиотечный модуль `signal` (см. главу 9).

Исключение `StopIteration` — часть протокола итераций. Оно сигнализирует о завершении итерации.

3.4.3. Определение новых исключений

Все встроенные исключения определяются в виде классов. Для нового исключения создайте новое определение класса, наследуемого от `Exception`, как в следующем примере:

```
class NetworkError(Exception):
    pass
```

Для использования нового исключения примените команду `raise`:

```
raise NetworkError('Cannot find host')
```

При выдаче исключения необязательные значения, переданные команде `raise`, используются как аргументы конструктора класса исключения. В большинстве случаев это строка с разновидностью сообщения об ошибке. Но исключения, определяемые пользователем, можно записать так, чтобы они получили одно или несколько значений:

```
class DeviceError(Exception):
    def __init__(self, errno, msg):
        self.args = (errno, msg)
        self.errno = errno
        self.errmsg = msg

# Выдает исключение (несколько аргументов)
raise DeviceError(1, 'Not Responding')
```

При создании специализированного класса исключения, который переопределяет `__init__()`, важно присвоить кортеж с аргументами `__init__()` атрибуту `self.args`, как показано в этом фрагменте. Атрибут используется при выводе сообщений трассировки исключений. Если оставить его неопределенным, пользователи не увидят никакой полезной информации об исключении при возникновении ошибки.

Исключения можно организовать в иерархию с помощью механизма наследования. Например, `NetworkError`, определенное выше, может стать базовым классом для более конкретных ошибок:

```
class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

def error1():
    raise HostnameError('Unknown host')

def error2():
    raise TimeoutError('Timed out')

try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # Выполнить действия для этой конкретной ошибки
    ...
```

Здесь секция `except NetworkError` перехватывает любые исключения, производные от `NetworkError`. Чтобы узнать конкретный тип выданного исключения, проверьте тип значения исключения функцией `type()`.

3.4.4. Цепочки исключений

Иногда в ответ на исключение нужно дать другое исключение. Для этого используются сцепленные исключения:

```
class ApplicationError(Exception):
    pass

def do_something():
    x = int('N/A') # Выдает ValueError

def spam():
    try:
        do_something()
    except Exception as e:
        raise ApplicationError('It failed') from e
```

При возникновении неперехваченного исключения `ApplicationError` вы получите сообщение с обоими исключениями:

```
>>> spam()
Traceback (most recent call last):
  File "c.py", line 9, in spam
    do_something()
  File "c.py", line 5, in do_something
    x = int('N/A')
ValueError: invalid literal for int() with base 10: 'N/A'
```

Это исключение было причиной следующего исключения:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c.py", line 11, in spam
    raise ApplicationError('It failed') from e
__main__.ApplicationError: It failed
>>>
```

Если вы перехватываете `ApplicationError`, атрибут `__cause__` полученного исключения будет содержать другое исключение:

```
try:
    spam()
except ApplicationError as e:
    print('It failed. Reason:', e.__cause__)
```

Если вы хотите выдать новое исключение без добавления цепочки других исключений, выдайте ошибку с указанием `from None`:

```
def spam():
    try:
        do_something()
    except Exception as e:
        raise ApplicationError('It failed') from None
```

Ошибка в блоке `except` тоже приведет к сцеплению исключений, но работает она несколько иначе. Представьте, что у вас есть код с ошибкой:

```
def spam():
    try:
        do_something()
    except Exception as e:
        print('It failed:', err) # err undefined (typo)
```

Итоговое сообщение трассировки исключения несколько изменяется:

```
>>> spam()
Traceback (most recent call last):
  File "d.py", line 9, in spam
```

```

    do_something()
File "d.py", line 5, in do_something
    x = int('N/A')
ValueError: invalid literal for int() with base 10: 'N/A'

```

During handling of the above exception, another exception occurred:

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "d.py", line 11, in spam
    print('It failed. Reason:', err)
NameError: name 'err' is not defined
>>>

```

Если при обработке другого исключения возникает неожиданное исключение, в атрибуте `__context__` (вместо `__cause__`) будет информация об исключении, которое обрабатывалось в момент возникновения ошибки:

```

try:
    spam()
except Exception as e:
    print('It failed. Reason:', e)
    if e.__context__:
        print('While handling:', e.__context__)

```

Между ожидаемыми и неожиданными исключениями в цепочках есть разница. В первом примере код был написан с предположением возможности исключения. Например, он был явно упакован в блок `try-except`:

```

try:
    do_something()
except Exception as e:
    raise ApplicationError('It failed') from e

```

Во втором случае программная ошибка возникает в блоке `except`:

```

try:
    do_something()
except Exception as e:
    print('It failed:', err) # Значение err не определено

```

Между этими двумя случаями есть тонкие, но важные различия. Именно поэтому информация о сцепленных исключениях помещается либо в атрибут `__cause__`, либо в `__context__`. Атрибут `__cause__` резервируется для ситуаций, где вы ожидаете возможную ошибку. Атрибут `__context__` устанавливается в обоих случаях, но становится единственным источником информации для непредвиденного исключения, выданного при обработке другого исключения.

3.4.5. Трассировка исключений

Исключения сопровождаются трассировкой стека, предоставляющей информацию о месте возникновения ошибки. Трассировка хранится в атрибуте `__traceback__` исключения. Возможно, вы захотите сами сформировать сообщение трассировки при составлении отчета или отладке. Для этого можно воспользоваться модулем `traceback`:

```
import traceback

try:
    spam()
except Exception as e:
    tblines = traceback.format_exception(type(e), e, e.__traceback__)
    tbmsg = ''.join(tblines)
    print('It failed:')
    print(tbmsg)
```

В этом коде `format_exception()` выдает список строк с выводом, который обычно генерируется Python в сообщении трассировки. На вход подается тип исключения, значение и трассировка.

3.4.6. Рекомендации по обработке ошибок

Обработку исключений сложнее всего правильно реализовать в больших программах. Но соблюдение некоторых правил упростит решение этой задачи.

Правило первое — не перехватывайте исключения, которые не могут быть обработаны в текущей позиции программы. Возьмем следующую функцию:

```
def read_data(filename):
    with open(filename, 'rt') as file:
        rows = []
        for line in file:
            row = line.split()
            rows.append((row[0], int(row[1]), float(row[2])))
    return rows
```

Допустим, вызов функции `open()` завершается неудачей из-за неверного имени файла. Та ли это ошибка, которая должна перехватываться командой `try-except` в этой функции? Вероятно, нет. Если сторона вызова передает недопустимое имя файла, разумного варианта восстановления нет. Нет файла, который можно было бы открыть, данных, которые можно было бы прочесть. Нет вообще ничего. Лучше разрешить ошибку при попытке выполнения операции и сообщить об исключении на сторону вызова. Отсутствие проверки ошибки в `read_data()` не означает, что исключение никогда не будет нигде

обработано. Это просто значит, что этим не должна заниматься функция `read_data()`. Возможно, исключение будет обработано кодом, который запрашивал имя файла у пользователя.

Этот совет может показаться противоречащим опыту программистов, привыкших к языкам, которые полагаются на специальные коды ошибок или обернутые типы результатов. В таких языках программист всегда должен действовать крайне осторожно и проверять коды ошибок для всех операций. В Python это необязательно. Если операция может завершиться неудачно, и вы ничего не можете сделать для восстановления, лучше дать ей завершиться неудачей. Исключение — более высокие уровни программы, где за обработку обычно отвечает другой код.

Может оказаться, что функция способна восстановиться от некорректных данных:

```
def read_data(filename):
    with open(filename, 'rt') as file:
        rows = []
        for line in file:
            row = line.split()
            try:
                rows.append((row[0], int(row[1]), float(row[2])))
            except ValueError as e:
                print('Bad row:', row)
                print('Reason:', e)
    return rows
```

Перехватывая ошибки, старайтесь делать секции `except` как можно точнее. Код выше можно было бы переписать так, чтобы в нем перехватывались все ошибки с помощью конструкции `except Exception`. Но тогда код будет перехватывать законные ошибки, которые, скорее всего, игнорироваться не должны. Не делайте этого — вы только усложните процесс отладки.

Наконец, если вы явно вызываете исключение, рассмотрите возможность создания собственных типов исключений. Пример:

```
class ApplicationError(Exception):
    pass

class UnauthorizedUserError(ApplicationError):
    pass

def spam():
    ...
    raise UnauthorizedUserError('Go away')
    ...
```

Одна из самых сложных проблем при работе с большими кодовыми базами — распределение ответственности за ошибки в программе. Если вы будете создавать свои исключения, вам будет проще отличать намеренно выданные ошибки от обоснованных ошибок программирования. Если в вашей программе возникает ошибка типа `ApplicationError`, вы немедленно узнаете о причине ее появления — потому что вы написали код, который ее выдает. С другой стороны, сбой в программе с одним из встроенных исключений Python (например, `TypeError` или `ValueError`) может указывать на более серьезную проблему.

3.5. МЕНЕДЖЕРЫ КОНТЕКСТА И КОМАНДА WITH

Управление системными ресурсами (файлами, блокировками и подключениями) часто становится серьезной проблемой в сочетании с исключениями. Из-за выданного исключения последовательность выполнения может обойти команды, ответственные за освобождение критических ресурсов (например, блокировок).

`with` позволяет выполнить набор команд в контексте среды выполнения под управлением объекта, выполняющего функции менеджера контекста:

```
with open('debuglog', 'wt') as file:
    file.write('Debugging\n')
    команды
    file.write('Done\n')

import threading
lock = threading.Lock()
with lock:
    # Критическая секция
    команды
    # Конец критической секции
```

В первом примере `with` автоматически вызывает закрытие открытого файла, когда поток управления покидает блок команд, следующих за ним. Во втором примере `with` автоматически захватывает и освобождает блокировку, когда управление входит и выходит из расположенного далее блока команд.

Команда `with obj` позволяет объекту `obj` управлять происходящим при входе и выходе управления из расположенного далее блока команд. При выполнении команды `with obj` вызывается метод `obj.__enter__()`. Он сигнализирует

о входе в новый контекст. При выходе управления из контекста выполняется метод `obj.__exit__(type, value, traceback)`. Если исключение не выдавалось, то всем трем аргументам `__exit__()` присваивается значение `None`. В противном случае в них содержатся тип, значение и трассировка, связанные с исключением, которое заставило управление покинуть контекст. Возвращение `True` методом `__exit__()` указывает на то, что выданное исключение было обработано и распространяться далее не должно. Возвращаемое значение `None` или `False` приводит к распространению исключения.

Команда `with obj` получает необязательный спецификатор `as var`. Если он задан, то значение, возвращенное `obj.__enter__()`, помещается в `var`. Обычно оно совпадает с `obj` — это позволяет сконструировать объект и использовать его как менеджер контекста за один шаг. Рассмотрим следующий класс:

```
class Manager:
    def __init__(self, x):
        self.x = x

    def yow(self):
        pass

    def __enter__(self):
        return self

    def __exit__(self, ty, val, tb):
        pass
```

С этим классом можно создать и использовать экземпляр менеджера контекста за один шаг:

```
with Manager(42) as m:
    m.yow()
```

Более интересный пример с использованием транзакций:

```
class ListTransaction:
    def __init__(self, thelist):
        self.thelist = thelist

    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy

    def __exit__(self, type, value, tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

Этот класс позволяет внести серию изменений в существующий список. Но они закрепляются только при отсутствии исключений. В противном случае исходный список остается без изменений:

```
items = [1,2,3]

with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items) # Выдает [1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("We're hosed!")
except RuntimeError:
    pass

print(items) # Выдает [1,2,3,4,5]
```

Модуль стандартной библиотеки `contextlib` содержит функциональные возможности, связанные с более продвинутым использованием менеджеров контекста. Если вам приходится регулярно создавать менеджеры контекста, познакомьтесь ближе с этим модулем.

3.6. КОМАНДЫ ASSERT И __DEBUG__

Команда `assert` может добавлять в программу отладочный код. Обобщенная форма `assert` выглядит так:

```
assert условие [, сообщение]
```

где условие — выражение, которое должно давать результат `True` или `False`. Если результат равен `False`, `assert` выдает исключение `AssertionError` с обязательным сообщением, переданным команде `assert`. Пример:

```
def write_data(file, data):
    assert file, 'write_data: file not defined!'
    ...
```

Не стоит использовать `assert` для кода, который должен выполняться для обеспечения правильности работы программы. Этот код не будет выполняться при выполнении Python в оптимизированном режиме (включается параметром `-O` при запуске интерпретатора). Неправильно использовать `assert`

и для проверки пользовательского ввода или успеха важной операции. Вместо этого команды `assert` должны применяться для проверки инвариантов, которые всегда должны быть истинными. Нарушение инварианта указывает на ошибку в программе, а не ошибку со стороны пользователя.

Например, если бы функция `write_data()` предназначалась для применения конечным пользователем, команду `assert` следовало бы заменить обычной командой `if` и необходимой обработкой исключений.

Обычно команды `assert` используются при тестировании. Можно использовать их для включения простейшей проверки функции:

```
def factorial(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result

assert factorial(5) == 120
```

Это неполная проверка. Она лишь служит тестом на общую работоспособность. Если в функции есть какая-то явная неисправность, то в программе немедленно произойдет сбой из-за непрошедшей проверки `assert` при импорте.

Команды `assert` могут пригодиться для определения программного контракта с ожидаемым вводом и выводом:

```
def factorial(n):
    assert n > 0, "must supply a positive value"
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

И здесь команда `assert` тоже не предназначена для проверки пользовательского ввода. Скорее, она проверяет внутреннюю целостность программы. Если какой-то другой код попытается вычислить факториал отрицательного числа, проверка завершится неудачей и укажет на код-нарушитель для отладки.

3.7. НАПОСЛЕДОК

Python поддерживает много разных стилей программирования, где задействованы функции и объекты. Но базовая модель выполнения программы — модель

императивного программирования. Другими словами, программы состоят из команд, выполняющихся одна за другой в том порядке, в котором они следуют в исходном файле. Есть только три основные конструкции управления последовательностью выполнения: команда `if`, команда `while` и команда `for`. Будет несложно разобраться, как Python выполняет вашу программу.

Конечно, сложнейшая и потенциально подверженная ошибкам область — это обработка исключений. Большая часть этой главы о том, как правильно рассуждать при организации обработки исключений. Даже если вы последуете нашим советам, исключения остаются чувствительной частью проектирования библиотек, фреймворков и API. Они могут вносить хаос и в управление ресурсами — эта задача решается использованием менеджера контекстов и командой `with`.

В этой главе не рассматриваются средства для настройки почти каждого аспекта языка Python — включая встроенные операторы и даже отдельные аспекты управления последовательностью выполнения. И хотя программы Python часто кажутся невероятно простыми по своей структуре, «за кулисами» происходит много невидимых событий. Некоторые описаны в следующей главе.

ГЛАВА 4

Объекты, типы и протоколы

Программы Python работают с объектами разных типов. Есть много встроенных типов: числа, строки, списки, множества, словари и т. д. Вы можете определять и свои типы с помощью классов. В этой главе описывается объектная модель Python и механизмы, обеспечивающие работу объектов. Особое внимание уделяется «протоколам», определяющим базовое поведение разных объектов.

4.1. ВАЖНЕЙШИЕ КОНЦЕПЦИИ

Каждый блок данных в программе — это объект. У каждого объекта есть идентификатор (*identity*), тип (еще называемый его классом) и значение. Например, при выполнении команды `a = 42` создается объект, представляющий целое число, со значением 42. Идентификатор объекта — это число, представляющее его местоположение в памяти, `a` — метка, относящаяся к этой конкретной ячейке памяти, хотя метка — не часть самого объекта.

Тип (класс) объекта определяет как внутренние данные объекта, так и поддерживаемые им методы. Объект конкретного типа называется *экземпляром* этого типа. После создания экземпляра его идентификатор не изменяется. Объект называется изменяемым, если его значение может быть изменено. Если значение изменяться не может, то объект называется неизменяемым. Объект, который хранит ссылки на другие объекты, называется контейнером.

Объекты характеризуются их атрибутами. Атрибут — это значение, связанное с объектом, для обращения к которому используется оператор «точка» (`.`). Атрибут может содержать простое значение данных, например число. Но он может содержать и функцию, которая вызывается для выполнения некоторой операции. Такие функции называются методами. Обращение с атрибутами иллюстрирует следующий пример:

```

a = 34          # Создать целое число
n = a.numerator # Получить numerator (атрибут)
b = [1, 2, 3]   # Создать список
b.append(7)     # Добавить новый элемент методом append

```

Объекты тоже могут реализовать разные операторы, например оператор +:

```

c = a + 10      # c = 34 + 10
d = b + [4, 5]  # d = [1, 2, 3, 7, 4, 5]

```

И хотя операторы используют другой синтаксис, в итоге они связываются с методами. Например, при вычислении выражения `a + 10` выполняется метод `a.__add__(10)`.

4.2. ИДЕНТИФИКАТОР ОБЪЕКТА И ЕГО ТИП

Встроенная функция `id()` возвращает идентификатор объекта. Это целое число, отображающее местонахождение объекта в памяти. Операторы `is` и `is not` сравнивают идентификаторы двух объектов. `type()` возвращает тип объекта. В следующем примере показаны разные способы сравнения двух объектов:

```

# Сравнение двух объектов
def compare(a, b):
    if a is b:
        print('same object')
    if a == b:
        print('same value')
    if type(a) is type(b):
        print('same type')

```

А вот как работает эта функция:

```

>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> compare(a, a)
same object
same value
same type
>>> compare(a, b)
same value
same type
>>> compare(a, [4,5,6])
same type
>>>

```

Тип объекта сам по себе является объектом, известным как класс объекта. Его определение неоднозначно и всегда остается неизменным для всех экземпляров заданного типа. У классов есть имена (`list`, `int`, `dict` и т. д.). Они используются для создания экземпляров, проверки типов, а также в аннотациях типов. Пример:

```
items = list()

if isinstance(items, list):
    items.append(item)
def removeall(items: list, item) -> list:
    return [i for i in items if i != item]
```

Подтип — это тип, определенный наследованием. Он сохраняет все возможности исходного типа и содержит дополнительные и/или переопределенные методы. Наследование подробнее рассматривается в главе 7, но ниже приведен пример определения подтипа списка, в который добавлен новый метод:

```
class mylist(list):
    def removeall(self, val):
        return [i for i in self if i != val]

# Пример
items = mylist([5, 8, 2, 7, 2, 13, 9])
x = items.removeall(2)
print(x)      # [5, 8, 7, 13, 9]
```

Проверить, что значение относится к заданному типу, можно с помощью функции `isinstance(экземпляр, тип)`. Она учитывает подтипы и поддерживает проверку по нескольким возможным типам. Пример:

```
if isinstance(items, (list, tuple)):
    maxval = max(items)
```

Проверки типов можно добавлять в программы, но часто они не сильно полезны. Во-первых, лишние проверки влияют на скорость. Во-вторых, программы не всегда определяют объекты, четко укладывающиеся в стройную иерархию типов.

Например, если цель оператора `isinstance(items, list)` проверить, являются ли элементы «спископодобными», он не будет работать с объектами, которые имеют тот же программный интерфейс, что и список, но не наследуются напрямую из встроенного типа списка (один из примеров — `deque` из модуля `collections`).

4.3. ПОДСЧЕТ ССЫЛОК И СБОР МУСОРА

Для управления объектами Python использует механизм автоматической сборки мусора. Для всех объектов ведутся счетчики ссылок. Счетчик ссылок увеличивается, когда объект связывается с новым именем или помещается в контейнер (список, кортеж или словарь):

```
a = 37          # Создает объект со значением 37
b = a           # Увеличивает счетчик ссылок для 37
c = []
c.append(b)     # Увеличивает счетчик ссылок для 37
```

Здесь создается один объект со значением 37. `a` — имя, которое изначально ссылается на созданный объект. Когда `a` присваивается `b`, `b` становится новым именем для того же объекта, а счетчик ссылок увеличивается. Когда `b` помещается в список, счетчик ссылок объекта снова увеличивается. Во всем коде примера значению 37 соответствует только один объект. Все остальные операции только создают новые ссылки на него.

Счетчик ссылок объекта уменьшается командой `del` и каждый раз, когда ссылка выходит из области видимости или когда переменной присваивается другая ссылка. Пример:

```
del a           # Уменьшает счетчик ссылок для 37
b = 42          # Уменьшает счетчик ссылок для 37
c[0] = 2.0     # Уменьшает счетчик ссылок для 37
```

Текущий счетчик ссылок объекта можно получить функцией `sys.getrefcount()`. Пример:

```
>>> a = 37
>>> import sys
>>> sys.getrefcount(a)
7
>>>
```

Значение счетчика ссылок часто оказывается намного выше ожидаемого. Для неизменяемых данных (числа и строки) интерпретатор активно обеспечивает совместное использование объектов между разными частями программы, чтобы сэкономить память. Вы просто не замечаете этого из-за неизменяемости объектов.

Когда счетчик ссылок достигает нуля, объект уничтожается сборщиком мусора. Но иногда в коллекциях неиспользуемых объектов могут появляться циклические ссылки:


```

a = { }
b = { }
a['b'] = b    # a содержит ссылку на b
b['a'] = a    # b содержит ссылку на a
del a
del b

```

Здесь команды `del` уменьшают счетчики ссылок `a` и `b` и уничтожают имена, используемые для ссылок на объекты ниже. Но каждый из этих объектов содержит ссылку на другой, поэтому счетчик ссылок не падает до нуля и объекты остаются в памяти. В интерпретаторе не снижается память, но уничтожение объектов откладывается до выполнения детектора циклов, занимающегося поиском и удалением недоступных объектов. Его алгоритм выполняется по мере того, как интерпретатор выделяет все больше памяти в ходе выполнения. Точное поведение этого алгоритма можно настраивать и управлять им при помощи функций из модуля стандартной библиотеки `gc`. Для немедленного запуска циклического сборщика мусора можно воспользоваться функцией `gc.collect()`.

В большинстве программ сборка мусора просто происходит сама по себе. Но иногда ручное удаление объектов может быть полезным, например при работе с гигантскими структурами данных. Рассмотрим следующий код:

```

def some_calculation():
    data = create_giant_data_structure()
    # data используется в некоторой части вычислений
    ...
    # Освобождение data
    del data

    # Вычисления продолжаются
    ...

```

Здесь команда `del data` сообщает, что переменная `data` больше не нужна. Если это приводит к уменьшению счетчика ссылок до 0, объект уничтожается сборщиком мусора. Без команды `del` объект продолжает существовать, пока переменная `data` не выйдет из области видимости в конце функции. Иногда это становится заметно только при попытке разобраться, почему программа расходует слишком много памяти.

4.4. ССЫЛКИ И КОПИИ

При выполнении присваивания в программе (например, `b = a`) создается новая ссылка на `a`. Может показаться, что для неизменяемых объектов (числа

и строки) оно создаст копию **a** (хотя это и не так). Но для изменяемых объектов (списков и словарей) поведение выглядит иначе:

```
>>> a = [1,2,3,4]
>>> b = a           # b содержит ссылку на a
>>> b is a
True
>>> b[2] = -100      # Изменяем элемент в b
>>> a                # Обратите внимание: содержимое a тоже изменилось
[1, 2, -100, 4]
>>>
```

В этом примере **a** и **b** ссылаются на один объект. Поэтому изменения, вносимые в одну переменную, отражаются на другой. Во избежание этого нужно создать копию объекта вместо новой ссылки.

К объектам-контейнерам (спискам и словарям) применяются операции копирования двух типов: поверхностного и глубокого. Поверхностное копирование создает новый объект, но заполняет его ссылками на элементы из исходного:

```
>>> a = [ 1, 2, [3,4] ]
>>> b = list(a)       # Создание поверхностной копии a
>>> b is a
False
>>> b.append(100)      # Присоединение элемента к b
>>> b
[1, 2, [3, 4], 100]
>>> a                 # Обратите внимание: a не изменяется
[1, 2, [3, 4]]
>>> b[2][0] = -100     # Изменение элемента внутри b
>>> b
[1, 2, [-100, 4], 100]
>>> a                 # Обратите внимание на изменения в a
[1, 2, [-100, 4]]
>>>
```

В этом случае **a** и **b** — разные объекты списков. Но элементы в них общие. Поэтому изменение элемента в **a** изменяет элемент и в **b**.

Глубокое копирование создает новый объект и рекурсивно копирует все содержащиеся в нем. Встроенного оператора для создания глубоких копий объектов нет, но вы можете воспользоваться функцией `copy.deepcopy()` из стандартной библиотеки:

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
```

```
>>> b
[1, 2, [-100, 4]]
>>> a                               # Обратите внимание: a не изменяется
[1, 2, [3, 4]]
>>>
```

В большинстве программ использовать `deepcopy()` не стоит. Копирование объектов — операция медленная и часто лишняя. Оставьте `deepcopy()` для ситуаций, где копирование действительно необходимо. Вы собираетесь изменять данные, но не хотите, чтобы изменения отразились на исходном объекте. Учтите, что `deepcopy()` не будет работать с объектами, в которых задействовано состояние системы или исполнительная среда (открытые файлы, сетевые подключения, программные потоки, генераторы и т. д.).

4.5. ПРЕДСТАВЛЕНИЕ И ВЫВОД ОБЪЕКТОВ

Программам часто приходится выводить объекты, например чтобы представить данные пользователю или вывести их для отладки. При передаче объекта `x` функции `print(x)` или преобразовании его в строку вызовом `str(x)` обычно вы получаете «красивое» представление значения объекта в удобном формате. Возьмем пример с использованием даты:

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> print(d)
2012-12-21
>>> str(d)
'2012-12-21'
>>>
```

«Красивого» представления объекта может быть недостаточно для отладки. Например, по выводу кода выше не удастся легко определить, является ли переменная `d` экземпляром `date` или простой строкой с текстом `'2012-12-21'`. Для получения дополнительной информации используйте функцию `repr(x)`. Она создает строку с представлением объекта, которое вам пришлось бы ввести в исходном коде для его создания. Пример:

```
>>> d = date(2012, 12, 21)
>>> repr(d)
'datetime.date(2012, 12, 21)'
>>> print(repr(d))
datetime.date(2012, 12, 21)
>>> print(f'The date is: {d!r}')
The date is: datetime.date(2012, 12, 21)
>>>
```

При форматировании строк к значению можно добавить суффикс `!r`, чтобы получить результат вызова `repr()` вместо нормального преобразования в строку.

4.6. ПЕРВОКЛАССНЫЕ ОБЪЕКТЫ

Все объекты в языке Python *первоклассные*. Это значит, что все объекты, которые могут быть связаны с именем, могут интерпретироваться и как данные. В качестве данных объекты могут сохраняться в переменных, передаваться в аргументах, возвращаться функциями, сравниваться с другими объектами и т. д. Следующий простой словарь содержит два значения:

```
items = {
    'number' : 42
    'text' : "Hello World"
}
```

Для демонстрации первоклассной природы объектов добавьте в словарь несколько необычных элементов:

```
items['func'] = abs           # Добавить функцию abs()
import math
items['mod'] = math           # Добавить модуль
items['error'] = ValueError   # Добавить тип исключения
nums = [1,2,3,4]
items['append'] = nums.append # Добавить метод другого объекта
```

Теперь словарь `items` содержит функцию, модуль, исключение и метод другого объекта. При желании можно использовать обращения к словарю `items` вместо исходных имен, и ваш код все равно будет работать:

```
>>> items['func'](-45)        # Выполнить abs(-45)
45
>>> items['mod'].sqrt(4)      # Выполнить math.sqrt(4)
2.0
>>> try:
...     x = int('a lot')
... except items['error'] as e: # Эквивалентно except ValueError as e
...     print("Couldn't convert")
...
Couldn't convert
>>> items['append'](100)      # Выполнить nums.append(100)
>>> nums
[1, 2, 3, 4, 100]
>>>
```

Начинающие программисты не сразу понимают важность первоклассности объектов. Но это позволяет писать очень компактный и гибкий код.

Представьте, что у вас есть строка текста (например, "АСМЕ,100,490.10"), которую нужно преобразовать в список значений с соответствующими преобразованиями типов. Следующий пример показывает возможное элегантное решение. Нужно создать список типов (являющихся первоклассными объектами) и выполнить несколько стандартных операций обработки списков:

```
>>> line = 'АСМЕ,100,490.10'
>>> column_types = [str, int, float]
>>> parts = line.split(',')
>>> row = [ty(val) for ty, val in zip(column_types, parts)]
>>> row
['АСМЕ', 100, 490.1]
>>>
```

Размещение функций или классов в словаре — стандартный прием для устранения сложных команд `if-elif-else`. Если у вас код следующего вида:

```
if format == 'text':
    formatter = TextFormatter()
elif format == 'csv':
    formatter = CSVFormatter()
elif format == 'html':
    formatter = HTMLFormatter()
else:
    raise RuntimeError('Bad format')
```

его можно переписать с использованием словаря:

```
_formats = {
    'text': TextFormatter,
    'csv': CSVFormatter,
    'html': HTMLFormatter
}

if format in _formats:
    formatter = _formats[format]()
else:
    raise RuntimeError('Bad format')
```

Вторая форма тоже довольно гибкая — для добавления новых случаев достаточно вставить в словарь новые элементы без изменения большого блока `if-elif-else`.

4.7. ИСПОЛЬЗОВАНИЕ NONE ДЛЯ НЕОБЯЗАТЕЛЬНЫХ ИЛИ ОТСУТСТВУЮЩИХ ДАННЫХ

Иногда в программах нужно представить необязательное или отсутствующее значение. `None` отлично подходит для этой цели. `None` возвращается функциями, которые не возвращают явное значение. `None` также часто используется как значение по умолчанию для необязательных аргументов, чтобы функция могла проверить, передала ли сторона вызова значение для этого аргумента. У `None` нет атрибутов, а в логических выражениях оно интерпретируется как `False`.

Во внутреннем представлении `None` хранится в виде одиночного значения (синглета), то есть в интерпретаторе возможно только одно значение `None`. Поэтому в стандартном способе проверки значения на равенство `None` используется оператор `is`:

```
if value is None:
    команды
    ...
```

Проверка на равенство `None` с использованием оператора `==` тоже работает. Но лучше так не поступать, средства проверки кода могут пометить такую ситуацию как ошибку стиля программирования.

4.8. ПРОТОКОЛЫ ОБЪЕКТОВ И АБСТРАКЦИИ ДАННЫХ

Многие возможности Python определяются *протоколами*. Возьмем следующую функцию:

```
def compute_cost(unit_price, num_units):
    return unit_price * num_units
```

А теперь задайте себе вопрос: какие входные значения допустимы? Ответ обманчиво прост — разрешено все! На первый взгляд все выглядит так, словно функция может применяться к числам:

```
>>> compute_cost(1.25, 50)
62.5
>>>
```

И действительно, все работает так, как и ожидалось. Но функция этим не ограничивается. Вы можете использовать специальные числа — дроби или `Decimal`:

```
>>> from fractions import Fraction
>>> compute_cost(Fraction(5, 4), 50)
Fraction(125, 2)
>>> from decimal import Decimal
>>> compute_cost(Decimal('1.25'), Decimal('50'))
Decimal('62.50')
>>>
```

Но и это не все — функция работает с массивами и другими сложными структурами из таких пакетов, как `numpy`:

```
>>> import numpy as np
>>> prices = np.array([1.25, 2.10, 3.05])
>>> units = np.array([50, 20, 25])
>>> compute_cost(prices, quantities)
array([62.5 , 42.  , 76.25])
>>>
```

Она даже может проявлять неожиданное поведение:

```
>>> compute_cost('a lot', 10)
'a lota lota lota lota lota lota lota lota lot'
>>>
```

Но с некоторыми комбинациями типов происходит ошибка:

```
>>> compute_cost(Fraction(5, 4), Decimal('50'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in compute_cost
TypeError: unsupported operand type(s) for *: 'Fraction' and 'decimal.
Decimal'
>>>
```

В отличие от компилятора статического языка, Python не проверяет правильность поведения программы заранее. Вместо этого поведение объекта определяется динамическим процессом, где задействованы так называемые специальные, или «волшебные», методы. Имена этих методов всегда начинаются и заканчиваются двойным подчеркиванием (`__`). Методы автоматически запускаются интерпретатором во время выполнения программы. Например, операция `x * y` выполняется методом `x.__mul__(y)`.

Имена методов и соответствующие им операторы жестко фиксированы. Поведение любого конкретного объекта полностью зависит от набора реализуемых им методов.

Следующие несколько разделов описывают специальные методы, связанные с разными категориями основных функций интерпретатора. Они часто называются протоколами. Любой объект (в том числе и пользовательский класс) может определить любую комбинацию этих возможностей для изменения поведения объекта.

4.9. ПРОТОКОЛ ОБЪЕКТОВ

Методы из табл. 4.1 относятся к общему управлению объектами: созданию объектов, инициализации, уничтожению и формированию представления.

Таблица 4.1. Методы управления объектами

| МЕТОД | ОПИСАНИЕ |
|--|---|
| <code>__new__(cls [, *args [, **kwargs]])</code> | Статический метод, вызываемый для создания нового экземпляра |
| <code>__init__(self [, *args [, **kwargs]])</code> | Вызывается для инициализации нового экземпляра после его создания |
| <code>__del__(self)</code> | Вызывается при уничтожении экземпляра |
| <code>__repr__(self)</code> | Создает строковое представление |

Методы `__new__()` и `__init__()` используются для создания и инициализации экземпляров. Когда объект создается вызовом `SomeClass(args)`, процесс создания можно описать следующим кодом:

```
x = SomeClass.__new__(SomeClass, args)
if isinstance(x, SomeClass):
    x.__init__(args)
```

Обычно эти шаги выполняются «за кулисами» и вам не приходится о них думать. Из всех методов классов чаще всего реализуется `__init__()`. Использование `__new__()` почти всегда свидетельствует о каких-то простых действиях, связанных с созданием экземпляра (например, он используется в методах классов, которые должны обходить `__init__()`, или в некоторых паттернах проектирования, относящихся к созданию объектов). Реализация `__new__()` не обязательно должна возвращать экземпляр класса. В этом случае последующий вызов `__init__()` при создании пропускается.

Метод `__del__()` вызывается, когда экземпляр должен быть уничтожен сборщиком мусора, и только в том случае, если объект больше не используется. Обратите внимание: команда `del x` только уменьшает счетчик ссылок и не обязательно приводит к вызову этой функции. `__del__()` почти никогда не определяется, если экземпляру не нужно выполнять дополнительные действия по управлению ресурсами после уничтожения.

Метод `__repr__()`, вызываемый встроенной функцией `repr()`, создает строковое представление объекта. Оно может быть полезно для отладки и вывода. Этот метод отвечает и за создание вывода, который вы получаете при проверке переменных в интерактивном интерпретаторе. По общепринятым соглашениям `__repr__()` возвращает строковое выражение, которое может быть вычислено для повторного создания объекта вызовом `eval()`:

```
a = [2, 3, 4, 5] # Создать список
s = repr(a)      # s = '[2, 3, 4, 5]'
b = eval(s)      # Преобразует s обратно в список
```

Если не удастся создать строковое выражение, по соглашениям `__repr__()` возвращает строку в форме `<...сообщение...>`:

```
f = open('foo.txt')
a = repr(f)
# a = "<_io.TextIOWrapper name='foo.txt' mode='r' encoding='UTF-8'"
```

4.10. ЧИСЛОВОЙ ПРОТОКОЛ

В табл. 4.2 перечислены специальные методы, которые объекты должны реализовывать для выполнения математических операций.

Таблица 4.2. Методы математических операций

| МЕТОД | ОПЕРАЦИЯ |
|--|--|
| <code>__add__(self, other)</code> | <code>self + other</code> |
| <code>__sub__(self, other)</code> | <code>self - other</code> |
| <code>__mul__(self, other)</code> | <code>self * other</code> |
| <code>__truediv__(self, other)</code> | <code>self / other</code> |
| <code>__floordiv__(self, other)</code> | <code>self // other</code> |
| <code>__mod__(self, other)</code> | <code>self % other</code> |
| <code>__matmul__(self, other)</code> | <code>self @ other</code> |
| <code>__divmod__(self, other)</code> | <code>divmod(self, other)</code> |
| <code>__pow__(self, other [, modulo])</code> | <code>self ** other, pow(self, other, modulo)</code> |

Таблица 4.2 (продолжение)

| МЕТОД | ОПЕРАЦИЯ |
|---|-----------------------------------|
| <code>__lshift__(self, other)</code> | <code>self << other</code> |
| <code>__rshift__(self, other)</code> | <code>self >> other</code> |
| <code>__and__(self, other)</code> | <code>self & other</code> |
| <code>__or__(self, other)</code> | <code>self other</code> |
| <code>__xor__(self, other)</code> | <code>self ^ other</code> |
| <code>__radd__(self, other)</code> | <code>other + self</code> |
| <code>__rsub__(self, other)</code> | <code>other - self</code> |
| <code>__rmul__(self, other)</code> | <code>other * self</code> |
| <code>__rtruediv__(self, other)</code> | <code>other / self</code> |
| <code>__rfloordiv__(self, other)</code> | <code>other // self</code> |
| <code>__rmod__(self, other)</code> | <code>other % self</code> |
| <code>__rmatmul__(self, other)</code> | <code>other @ self</code> |
| <code>__rdivmod__(self, other)</code> | <code>divmod(other, self)</code> |
| <code>__rpow__(self, other)</code> | <code>other ** self</code> |
| <code>__rlshift__(self, other)</code> | <code>other << self</code> |
| <code>__rrshift__(self, other)</code> | <code>other >> self</code> |
| <code>__rand__(self, other)</code> | <code>other & self</code> |
| <code>__ror__(self, other)</code> | <code>other self</code> |
| <code>__rxor__(self, other)</code> | <code>other ^ self</code> |
| <code>__iadd__(self, other)</code> | <code>self += other</code> |
| <code>__isub__(self, other)</code> | <code>self -= other</code> |
| <code>__imul__(self, other)</code> | <code>self *= other</code> |
| <code>__itruediv__(self, other)</code> | <code>self /= other</code> |
| <code>__ifloordiv__(self, other)</code> | <code>self //= other</code> |
| <code>__imod__(self, other)</code> | <code>self %= other</code> |
| <code>__imatmul__(self, other)</code> | <code>self @= other</code> |
| <code>__ipow__(self, other)</code> | <code>self **= other</code> |
| <code>__iand__(self, other)</code> | <code>self &= other</code> |
| <code>__ior__(self, other)</code> | <code>self = other</code> |
| <code>__ixor__(self, other)</code> | <code>self ^= other</code> |
| <code>__ilshift__(self, other)</code> | <code>self <<= other</code> |

| МЕТОД | ОПЕРАЦИЯ |
|---------------------------------------|-----------------------------------|
| <code>__irshift__(self, other)</code> | <code>self >>= other</code> |
| <code>__neg__(self)</code> | <code>-self</code> |
| <code>__pos__(self)</code> | <code>+self</code> |
| <code>__invert__(self)</code> | <code>~self</code> |
| <code>__abs__(self)</code> | <code>abs(self)</code> |
| <code>__round__(self, n)</code> | <code>round(self, n)</code> |
| <code>__floor__(self)</code> | <code>math.floor(self)</code> |
| <code>__ceil__(self)</code> | <code>math.ceil(self)</code> |
| <code>__trunc__(self)</code> | <code>math.trunc(self)</code> |

Сталкиваясь с выражением (например, $x + y$), интерпретатор вызывает комбинацию методов `x.__add__(y)` или `y.__radd__(x)` для выполнения операции. Сначала он пытается применять `x.__add__(y)` во всех случаях, кроме особого, когда `y` оказывается подтипом `x`, — тогда сначала выполняется `y.__radd__(x)`. Если попытка вызова исходного метода завершается ошибкой с возвращением `NotImplemented`, совершается попытка вызова операции с переставленными операндами, например `y.__radd__(x)`. Если и вторая попытка не проходит, то вся операция завершается неудачей:

```
>>> a = 42          # int
>>> b = 3.7         # float
>>> a.__add__(b)
NotImplemented
>>> b.__radd__(a)
45.7
>>>
```

Этот пример может показаться удивительным, но он показывает, что целые числа ничего не знают о числах с плавающей точкой. При этом числа с плавающей точкой знают о целых числах. Ведь с математических позиций целые числа — это особый случай чисел с плавающей точкой. Поэтому перестановка операндов дает правильный ответ.

Методы `__iadd__()`, `__isub__()` и т. д. используются для поддержания таких арифметических операций, как `a += b` или `a -= b` (еще их называют комбинированным присваиванием), «на месте». Эти операторы отличаются от стандартных арифметических методов. Реализация операторов присваивания «на месте» может предоставить некоторые настройки или оптимизации скорости. Например, если объект не используется совместно, его значение может быть изменено на месте без выделения вновь созданного объекта для результата.

Если операторы присваивания «на месте» остаются неопределенными, такие операции, как `a += b`, будут вычисляться в виде `a = a + b`.

Нет методов, которые могли бы использоваться для определения поведения логических операторов `and`, `or` и `not`. Операторы `and` и `or` реализуют ускоренное вычисление, которое останавливается, когда уже можно определить окончательный результат:

```
>>> True or 1/0 # Не вычисляет 1/0
True
>>>
```

Это поведение, где задействованы невычисляемые подвыражения, не может быть выражено правилами вычисления обычных функций или методов. Поэтому нет протокола или множества методов для его переопределения. Вместо этого он обрабатывается как специальный случай глубоко внутри самой реализации Python.

4.11. ПРОТОКОЛ СРАВНЕНИЯ

Объекты могут сравниваться по-разному. Простейший способ сравнения — проверка тождественности оператором `is` (например `a is b`). Тождественность не учитывает значения внутри объекта, даже если они окажутся одинаковыми:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> c = [1, 2, 3]
>>> a is c
False
>>>
```

Оператор `is` — внутренняя часть Python, которая не может переопределяться. Все остальные сравнения объектов реализуются методами из табл. 4.3.

Метод `__bool__()` используется для определения истинности, когда объект проверяется как часть условия или условного выражения:

```
if a: # Выполняет a.__bool__()
    ...
else:
    ...
```

Если значение `__bool__()` не определено, то `__len__()` используется в качестве резервного критерия. Если и `__bool__()`, и `__len__()` не определены, объект просто считается интерпретируемым как `True`.

Таблица 4.3. Методы сравнения экземпляров и хеширования

| МЕТОД | ОПИСАНИЕ |
|----------------------------------|--|
| <code>__bool__(self)</code> | Возвращает <code>False</code> или <code>True</code> для проверки истинности значения |
| <code>__eq__(self, other)</code> | <code>self == other</code> |
| <code>__ne__(self, other)</code> | <code>self != other</code> |
| <code>__lt__(self, other)</code> | <code>self < other</code> |
| <code>__le__(self, other)</code> | <code>self <= other</code> |
| <code>__gt__(self, other)</code> | <code>self > other</code> |
| <code>__ge__(self, other)</code> | <code>self >= other</code> |
| <code>__hash__(self)</code> | Вычисляет целочисленный хеш-код |

`__eq__()` предназначено для того, чтобы определить базовое равенство для использования с операторами `==` и `!=`. Реализация `__eq__()` по умолчанию сравнивает объекты по тождественности с помощью `is`. Метод `__ne__()` может использоваться для выполнения специальной обработки `!=`. Но обычно это не нужно, если определен метод `__eq__()`.

Упорядочение определяется операторами отношений (`<`, `>`, `<=` и `>=`) с использованием таких методов, как `__lt__()` и `__gt__()`. Как и в случае с другими математическими операторами, в правилах вычисления есть нюансы. Чтобы вычислить `a < b`, интерпретатор сначала пытается выполнить `a.__lt__(b)` — кроме случаев, где `b` — подтип `a`. В этом конкретном случае выполняется `b.__gt__(a)`.

Если исходный метод не определен или возвращает `NotImplemented`, интерпретатор пытается выполнить обратное сравнение, вызывая `b.__gt__(a)`. Те же правила применяются к таким операторам, как `<=` и `>=`. Например, при вычислении `<=` интерпретатор сначала пытается вычислить `a.__le__(b)`. Если операция не реализована, делается попытка вычислить `b.__ge__(a)`.

Каждый метод сравнения получает два аргумента и может возвращать любое значение, включая логическое значение, список или другой тип Python. Вычислительный пакет может воспользоваться этой возможностью для выполнения поэлементных сравнений двух матриц и возвращения матрицы результатов. Если сравнение невозможно, методы должны возвращать встроенный

объект `NotImplemented` — не путайте его с исключением `NotImplementedError`.
Пример:

```
>>> a = 42    # int
>>> b = 52.3 # float
>>> a.__lt__(b)
NotImplemented
>>> b.__gt__(a)
True
>>>
```

Упорядоченные объекты не обязаны реализовать все операторы сравнения из табл. 4.3. Чтобы сортировать объекты или использовать такие функции, как `min()` или `max()`, нужно как минимум определить `__lt__()`. Если вы добавляете операторы сравнения к пользовательскому классу, вам пригодится декоратор класса `@total_ordering` из модуля `functools`. Он может сгенерировать все методы, при условии что вы реализовали как минимум `__eq__()`, и одно из других сравнений.

Метод `__hash__()` определяется для экземпляров, которые должны включаться в множества или использоваться как ключи в отображениях (словарях). Возвращаемое целое число должно быть одинаковым для двух экземпляров, признанных равными при сравнении. Метод `__eq__()` всегда должен определяться вместе с `__hash__()` — они работают совместно. Значение, возвращаемое `__hash__()`, обычно используется как внутренняя подробность реализации разных структур данных. Но у двух разных объектов могут быть одинаковые хеш-коды. Поэтому вызов `__eq__()` необходим для разрешения возможных коллизий.

4.12. ПРОТОКОЛЫ ПРЕОБРАЗОВАНИЯ

Иногда нужно преобразовать объекты во встроенные типы (строки или числа). Для этого определяются методы из табл. 4.4.

Метод `__str__()` вызывается встроенной функцией `str()` и функциями, связанными с выводом. Метод `__format__()` вызывается функцией `format()` или методом `format()` для строк. Аргумент `format_spec` содержит строку со спецификацией формата. Это такая же строка, как и та, что передается в аргументе `format_spec` функции `format()`. Пример:

```
f'{x:spec}'                            # Вызывает x.__format__('spec')
format(x, 'spec')                    # Вызывает x.__format__('spec')
'x is {0:spec}'.format(x)            # Вызывает x.__format__('spec')
```

У спецификации формата произвольный синтаксис, который может настраиваться на уровне отдельных объектов. Но есть стандартный набор соглашений для встроенных типов. Дополнительную информацию о форматировании строк, включая общий формат спецификаторов, см. в главе 9.

Таблица 4.4. Методы преобразований

| МЕТОД | ОПИСАНИЕ |
|--|---|
| <code>__str__(self)</code> | Преобразование в строку |
| <code>__bytes__(self)</code> | Преобразование в последовательность байтов |
| <code>__format__(self, format_spec)</code> | Создает отформатированное представление |
| <code>__bool__(self)</code> | <code>bool(self)</code> |
| <code>__int__(self)</code> | <code>int(self)</code> |
| <code>__float__(self)</code> | <code>float(self)</code> |
| <code>__complex__(self)</code> | <code>complex(self)</code> |
| <code>__index__(self)</code> | Преобразование в целочисленный индекс <code>[self]</code> |

Метод `__bytes__()` используется для создания байтового представления экземпляра, переданного `bytes()`. Не все типы поддерживают преобразование в байты.

Предполагается, что числовые преобразования `__bool__()`, `__int__()`, `__float__()` и `__complex__()` производят значение соответствующего встроенного типа.

Python никогда не выполняет неявные преобразования типов с использованием этих методов. Даже если объект `x` реализует метод `__int__()`, выражение `3 + x` все равно приведет к ошибке `TypeError`. Метод `__int__()` может быть выполнен только явным вызовом функции `int()`.

Метод `__index__()` преобразует объект в целое число, когда используется в операции, требующей целочисленного значения. К этой категории относится и индексирование в операциях с последовательностями.

Например, если `items` является списком, при выполнении такой операции, как `items[x]`, будет сделана попытка выполнить `items[x.__index__()]`, если `x` — не целое число. `__index__()` тоже используется в разных преобразованиях систем счисления, включая `oct(x)` и `hex(x)`.

4.13. ПРОТОКОЛ КОНТЕЙНЕРА

Методы из табл. 4.5 используются объектами, реализующими контейнеры разных типов — списки, словари, множества и т. д.

Таблица 4.5. Методы контейнеров

| МЕТОД | ОПИСАНИЕ |
|--|--|
| <code>__len__(self)</code> | Возвращает длину <code>self</code> |
| <code>__getitem__(self, key)</code> | Возвращает <code>self[key]</code> |
| <code>__setitem__(self, key, value)</code> | Присваивает <code>self[key] = value</code> |
| <code>__delitem__(self, key)</code> | Удаляет <code>self[key]</code> |
| <code>__contains__(self, obj)</code> | Проверяет наличие <code>obj</code> в <code>self</code> |

Пример:

```
a = [1, 2, 3, 4, 5, 6]
len(a)           # a.__len__()
x = a[2]         # x = a.__getitem__(2)
a[1] = 7         # a.__setitem__(1,7)
del a[2]         # a.__delitem__(2)
5 in a           # a.__contains__(5)
```

Метод `__len__()` вызывается встроенной функцией `len()` для возврата неотрицательной длины. Эта функция определяет истинность, если не был определен метод `__bool__()`.

При обращении к отдельным элементам метод `__getitem__()` может вернуть элемент по значению ключа. Ключом может быть любой объект Python, но ожидается, что это целое число для упорядоченных последовательностей (списков и массивов). Метод `__setitem__()` присваивает значение элементу. Метод `__delitem__()` вызывается, когда операция `del` применяется к одному элементу. Метод `__contains__()` используется для реализации оператора `in`.

Операции с сегментами, `x = s[i:j]`, тоже реализуются методами `__getitem__()`, `__setitem__()` и `__delitem__()`. Для сегментов в качестве ключа передается специальный экземпляр `slice`. Он содержит атрибуты, описывающие диапазон запрашиваемого сегмента:

```
a = [1,2,3,4,5,6]
x = a[1:5]       # x = a.__getitem__(slice(1, 5, None))
a[1:3] = [10,11,12] # a.__setitem__(slice(1, 3, None), [10, 11, 12])
del a[1:4]       # a.__delitem__(slice(1, 4, None))
```


Средства сегментации Python мощнее, чем многие думают. Например, поддерживаются следующие разновидности расширенной сегментации, которые могут быть полезны при работе с многомерными структурами данных (матрицами и массивами):

```
a = m[0:100:10]      # Сегмент с приращением (stride=10)
b = m[1:10, 3:20]    # Многомерный сегмент
c = m[0:100:10, 50:75:5] # Несколько размерностей с приращениями
m[0:5, 5:10] = n      # Расширенное присваивание сегмента
del m[:10, 15:]       # Расширенное удаление сегмента
```

Обобщенный формат каждого измерения расширенного сегмента имеет вид `i:j[:stride]`, где значение `stride` необязательно. Как и в случае с обычными сегментами, вы можете опустить начальные и конечные значения для каждой части сегмента.

Кроме того, любое число измерений в начале или в конце расширенного сегмента может обозначаться объектом `Ellipsis` (записывается как многоточие):

```
a = m[..., 10:20] # Обращение к расширенному сегменту
m[10:20, ...] = n
```

При использовании расширенных сегментов методы `__getitem__()`, `__setitem__()` и `__delitem__()` реализуют обращение, изменение и удаление. Но значение, передаваемое этим методам, вместо целого числа представляет комбинацию сегментов или объектов `Ellipsis`. Например,

```
a = m[0:10, 0:100:5, ...]
```

вызывает `__getitem__()` в следующем виде:

```
a = m.__getitem__((slice(0,10,None), slice(0,100,5), Ellipsis))
```

Строки, кортежи и списки Python сегодня предоставляют частичную поддержку расширенных сегментов. Ни в одной части Python или его стандартной библиотеки не используются многомерные сегменты или `Ellipsis`. Эти возможности зарезервированы только для сторонних библиотек и фреймворков. Чаще всего они используются в библиотеках (например `numpy`).

4.14. ПРОТОКОЛ ИТЕРАЦИЙ

Если экземпляр `obj` поддерживает перебор, он предоставляет метод `obj.__iter__()`, возвращающий итератор. Итератор `iter` реализует один метод `iter.__next__()`, который возвращает следующий объект или выдает

исключение `StopIteration` для обозначения конца перебора. Эти методы используются реализацией команды `for` и другими операциями, которые неявно выполняют перебор. Например, команда `for x in s` выполняется так:

```
_iter = s.__iter__()
while True:
    try:
        x = _iter.__next__()
    except StopIteration:
        break
    # Выполнить команды в теле цикла for
...

```

Объект может дополнительно предоставить обратный итератор, если он реализует специальный метод `__reversed__()`. Он должен возвращать объект итератора с таким же интерфейсом, как у обычного (то есть методом `__next__()`, выдающим `StopIteration` в конце перебора). Этот метод используется встроенной функцией `reversed()`. Пример:

```
>>> for x in reversed([1,2,3]):
...     print(x)
3
2
1
>>>

```

В стандартной реализации перебора используется функция-генератор, связанная с `yield`:

```
class FRange:
    def __init__(self, start, stop, step):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        x = self.start
        while x < self.stop:
            yield x
            x += self.step

# Пример использования:
nums = FRange(0.0, 1.0, 0.1)
for x in nums:
    print(x) # 0.0, 0.1, 0.2, 0.3, ...

```

Это решение работает, потому что функции-генераторы сами соответствуют протоколу итераций. Такая реализация итератора немного проще, ведь вам

придется беспокоиться только о методе `__iter__()`. Остальные части механизмов перебора уже предоставляются генератором.

4.15. ПРОТОКОЛ АТРИБУТОВ

Методы из табл. 4.6 читают, записывают и удаляют атрибуты объекта с использованием оператора «точка» (`.`) и `del`.

Таблица 4.6. Методы обращений к атрибутам

| МЕТОД | ОПИСАНИЕ |
|---|---|
| <code>__getattribute__(self, name)</code> | Возвращает атрибут <code>self.name</code> |
| <code>__getattr__(self, name)</code> | Возвращает атрибут <code>self.name</code> , если его не удалось найти вызовом <code>__getattribute__()</code> |
| <code>__setattr__(self, name, value)</code> | Присваивает значение атрибута <code>self.name = value</code> |
| <code>__delattr__(self, name)</code> | Удаляет атрибут <code>del self.name</code> |

При каждом обращении к атрибуту вызывается метод `__getattribute__()`. Если атрибут найден, его значение возвращается. В противном случае вызывается метод `__getattr__()`. По умолчанию `__getattr__()` выдает исключение `AttributeError`. Метод `__setattr__()` всегда вызывается при присвоении значения атрибуту, а `__delattr__()` — при удалении атрибута.

Эти методы довольно прямолинейны — они позволяют типу полностью перепреопределить доступ к атрибутам для всех атрибутов. Пользовательские классы могут определять свойства и дескрипторы, позволяющие точнее управлять обращениями к атрибутам. Эта тема подробнее рассматривается в главе 7.

4.16. ПРОТОКОЛ ФУНКЦИЙ

Объект может эмулировать функцию, предоставляя метод `__call__()`. Если объект `x` предоставляет этот метод, он может вызываться как функция. Иначе говоря, для `x(arg1, arg2, ...)` вызывается `x.__call__(arg1, arg2, ...)`.

Есть много встроенных типов, поддерживающих вызовы функций. Например, типы реализуют `__call__()` для создания новых экземпляров. Связанные методы реализуют `__call__()` для передачи аргумента `self` методам экземпляров. Библиотечные функции (`functools.partial()`) создают объекты, эмулирующие функции.

4.17. ПРОТОКОЛ МЕНЕДЖЕРА КОНТЕКСТА

`with` позволяет выполнить последовательность команд под управлением экземпляра, именуемого менеджером контекста. Обобщенный синтаксис выглядит так:

```
with context [ as var]:  
    команды
```

Предполагается, что объект контекста реализует методы из табл. 4.7.

Таблица 4.7. Методы менеджеров контекстов

| МЕТОД | ОПИСАНИЕ |
|--|--|
| <code>__enter__(self)</code> | Вызывается при входе в новый контекст. Возвращаемое значение помещается в переменную, указанную со спецификатором <code>as</code> в команде <code>with</code> |
| <code>__exit__(self, type, value, tb)</code> | Вызывается при выходе из контекста. Если произошло исключение, <code>type</code> , <code>value</code> и <code>tb</code> содержат тип исключения, его значение и информацию трассировки |

Метод `__enter__()` вызывается при выполнении команды `with`. Возвращаемое им значение помещается в переменную, заданную необязательным спецификатором `as var`. Метод `__exit__()` вызывается сразу же при выходе управления из блока команд, связанного с `with`. В аргументах `__exit__()` получает тип текущего исключения, значение и трассировку, если было выдано исключение. При отсутствии обрабатываемых ошибок все три значения содержат `None`. Метод `__exit__()` должен возвращать `True` или `False`, чтобы показать, было ли обработано исключение. Если возвращается `True`, все незавершенные исключения сбрасываются, а программа продолжает выполняться нормально с первой команды после блока `with`.

Основная задача интерфейса управления контекстом — упростить управление ресурсами для объектов, связанных с состоянием системы: открытых файлов, сетевых подключений и блокировок и т. д. Реализуя этот интерфейс, объект может безопасно освобождать ресурсы при выходе управления из контекста использования объекта. За дополнительной информацией обращайтесь к главе 3.

4.18. НАПОСЛЕДОК: О КОДЕ PYTHON

Среди целей проектирования часто упоминается написание кода Python. У этого термина много значений. Но по сути он рекомендует применять установившиеся идиомы, используемые сообществом Python. Это значит, что вы должны знать протоколы Python для контейнеров, итерируемых объектов, управления ресурсами и т. д. Многие популярные фреймворки Python используют эти протоколы для обеспечения хорошего опыта взаимодействия с пользователями. Вы тоже должны стремиться к этому.

Из многих протоколов три заслуживают особого внимания из-за своего широкого распространения. Один создает представление объекта методом `__repr__()`. Отладка и эксперименты с программами Python часто выполняются в интерактивной оболочке REPL. Этот протокол часто используется при выводе объектов функцией `print()` или библиотекой ведения журналов. Упростив наблюдение за состоянием своих объектов, вы упростите все перечисленные операции.

Далее, к числу самых распространенных задач в программировании принадлежит перебор данных. Если вы пойдете по этому пути, ваш код будет работать с командой `for` языка Python. Многие базовые части Python и стандартная библиотека разрабатывались для взаимодействия с итерируемыми объектами. Поддерживая итерации стандартным способом, вы автоматически получаете доступ к большей части дополнительной функциональности, и ваш код становится интуитивно понятным для других программистов.

Наконец, менеджеры контекста и команда `with` используются для реализации популярного программного паттерна, где команды заключаются между начальным и завершающим этапом: открытием и закрытием ресурсов, захватом и освобождением блокировки, подпиской и ее отменой, и т. д.

ГЛАВА 5

Функции

Функции — базовые структурные элементы большинства программ Python. В этой главе рассматриваются определения функций, их применение, правила области видимости, замыкания, декораторы и другие средства функционального программирования. Особое внимание уделяется разным идиомам программирования, моделям вычисления и паттернам, связанным с функциями.

5.1. ОПРЕДЕЛЕНИЯ ФУНКЦИЙ

Функции определяются командой `def`:

```
def add(x, y):  
    return x + y
```

Первая часть определения функции задает ее имя и имена параметров, определяющих входные значения. Тело функции содержит последовательность команд, выполняемых при ее вызове. Для вызова функции с заданными аргументами запишите имя функции, за которым следуют аргументы в круглых скобках: `a = add(3, 4)`. Аргументы полностью вычисляются слева направо перед выполнением тела функции. Например, `add(1+1, 2+2)` сначала редуцируется до `add(2, 4)` перед вызовом функции. Это называется *аппликативным порядком вычисления*. Порядок и число аргументов должны соответствовать параметрам, указанным в определении функции. При обнаружении несоответствия выдается исключение `TypeError`. Структура вызова функции (например, количество обязательных аргументов) называется сигнатурой вызова.

5.2. АРГУМЕНТЫ ПО УМОЛЧАНИЮ

Для параметров функций можно определить значения по умолчанию. Они указываются в определении функции:

```
def split(line, delimiter=','):
    команды
```

Когда функция определяет параметр со значением по умолчанию, этот и все последующие параметры необязательны. После параметра со значением по умолчанию нельзя задавать другие без этого значения.

Значения параметров по умолчанию вычисляются только при определении функции, а не при каждом ее вызове. Если по умолчанию используются изменяемые объекты, это может привести к неожиданному поведению:

```
def func(x, items=[]):
    items.append(x)
    return items

func(1)      # Возвращает [1]
func(2)      # Возвращает [1, 2]
func(3)      # Возвращает [1, 2, 3]
```

Обратите внимание: аргумент по умолчанию сохраняет изменения, внесенные при предыдущих вызовах. Для предотвращения таких ситуаций лучше использовать `None` и организовать проверку так:

```
def func(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

Обычно для предотвращения таких сюрпризов в качестве значений аргументов по умолчанию лучше использовать только неизменяемые значения — числа, строки, логические значения, `None` и т. д.

5.3. ФУНКЦИИ С ПЕРЕМЕННЫМ КОЛИЧЕСТВОМ АРГУМЕНТОВ

Функция может получать переменное количество аргументов. Для этого перед именем последнего параметра ставится префикс `*` (звездочка). Пример:

```
def product(first, *args):
    result = first
    for x in args:
        result = result * x
    return result

product(10, 20) # -> 200
product(2, 3, 4, 5) # -> 120
```

В этом случае все дополнительные аргументы помещаются в переменную `args` в виде кортежа. После этого с аргументами можно работать, используя стандартные операции последовательностей — перебор, сегментацию, распаковку и т. д.

5.4. КЛЮЧЕВЫЕ АРГУМЕНТЫ

При передаче аргументов функциям можно явно указать имена всех параметров вместе со значениями. Такие аргументы называются ключевыми:

```
def func(w, x, y, z):
    команды

# Вызов с ключевыми аргументами
func(x=3, y=22, w='hello', z=[1, 2])
```

С ключевыми аргументами порядок передачи неважен, если для каждого обязательного параметра передается одно значение. В случае отсутствия какого-либо из обязательных аргументов или несоответствия имени ключевого параметра ни одному имени параметра в определении функции выдается исключение `TypeError`. Ключевые аргументы вычисляются в том порядке, в котором они указываются при вызове функции.

Позиционные и ключевые аргументы могут сочетаться в одном вызове функции, если сначала перечисляются все позиционные аргументы, для всех обязательных аргументов указаны значения, и ни одному аргументу не присваивается более одного значения.

Пример:

```
func('hello', 3, z=[1, 2], y=22)
func(3, 22, w='hello', z=[1, 2]) # TypeError. Несколько значений для w
```

Можно включить принудительное использование ключевых аргументов. Для этого параметры перечисляются после аргумента `*` либо простым включением единственного символа `*` в определение:

```
def read_data(filename, *, debug=False):
    ...

def product(first, *values, scale=1):
    result = first * scale
    for val in values:
        result = result * val
    return result
```


Здесь аргумент `debug` функции `read_data()` может задаваться только как ключевой. Это ограничение часто улучшает удобочитаемость кода:

```
data = read_data('Data.csv', True)          # НЕТ. TypeError
data = read_data('Data.csv', debug=True)     # Да.
```

Функция `product()` получает любое число позиционных аргументов и обязательный аргумент, который может быть только ключевым:

```
result = product(2,3,4)                     # result = 24
result = product(2,3,4, scale=10)           # result = 240
```

5.5. ФУНКЦИИ С ПЕРЕМЕННЫМ ЧИСЛОМ КЛЮЧЕВЫХ АРГУМЕНТОВ

Если у последнего аргумента в определении функции есть префикс `**`, все дополнительные ключевые аргументы (не соответствующие ни одному имени параметра) помещаются в словарь и передаются функции. Порядок элементов этого словаря гарантированно совпадает с порядком передачи ключевых аргументов.

Произвольные ключевые аргументы могут пригодиться при определении функций, получающих большой (теоретически расширяемый) набор параметров конфигурации, которые было бы слишком неудобно перечислять в параметрах. Пример:

```
def make_table(data, **parms):
    # Получить параметры конфигурации из parms (словарь)
    fgcolor = parms.pop('fgcolor', 'black')
    bgcolor = parms.pop('bgcolor', 'white')
    width = parms.pop('width', None)
    ...
    # Параметров больше нет
    if parms:
        raise TypeError(f'Unsupported configuration options {list(parms)}')

make_table(items, fgcolor='black', bgcolor='white', border=1,
            borderstyle='grooved', cellpadding=10,
            width=400)
```

Метод `pop()` удаляет элемент из словаря, возвращая возможное значение по умолчанию, если оно не определено. Выражение `parms.pop('fgcolor', 'black')`, использованное в коде, имитирует поведение ключевого аргумента, заданного со значением по умолчанию.

5.6. ФУНКЦИИ, ПРИНИМАЮЩИЕ ЛЮБОЙ ВВОД

С помощью `*` и `**` можно написать функцию, которая принимает произвольную комбинацию аргументов. Позиционные аргументы передаются в кортеже, а ключевые — в словаре:

```
# Получает переменное количество позиционных или ключевых аргументов
def func(*args, **kwargs):
    # args — кортеж позиционных аргументов
    # kwargs — словарь ключевых аргументов
    ...
```

Используя как `*`, так и `**`, вы можете написать функцию, принимающую любую комбинацию аргументов. Представьте, что у вас есть функция для разбора строк текста, полученных от итерируемого объекта:

```
def parse_lines(lines, separator=',', types=(), debug=False):
    for line in lines:
        ...
        команды
        ...
```

Допустим, вы хотите написать специализированную функцию, разбирающую данные из файла, заданного по имени. Это можно сделать так:

```
def parse_file(filename, *args, **kwargs):
    with open(filename, 'rt') as file:
        return parse_lines(file, *args, **kwargs)
```

Преимущество такого подхода в том, что функции `parse_file()` не нужно ничего знать об аргументах `parse_lines()`. Она получает любые дополнительные аргументы строки вызова и передает их дальше. Такой подход упрощает сопровождение функции `parse_file()`. Например, если в `parse_lines()` будут добавлены новые аргументы, они как по волшебству будут работать и с функцией `parse_file()`.

5.7. ТОЛЬКО ПОЗИЦИОННЫЕ АРГУМЕНТЫ

Многие встроенные функции Python получают аргументы только в позиционном виде. На это указывает наличие косой черты (`/`) в сигнатуре вызова функции, показанной разными справочными утилитами и IDE, — что-то вроде `func(x, y, /)`. Это значит, что все аргументы до этой черты

могут задаваться только позицией. Поэтому функция может вызываться в виде `func(2, 3)`, но не `func(x=2, y=3)`. Для полноты этот синтаксис может использоваться и при определении функций. Можно использовать запись следующего вида:

```
def func(x, y, /):  
    pass  
  
func(1, 2)      # Ok  
func(1, y=2)    # Ошибка
```

Эта форма определения редко встречается в коде — она поддерживается, только начиная с Python 3.8. Но она может помочь предотвратить возможные конфликты имен аргументов. Для примера возьмем следующий фрагмент:

```
import time  
  
def after(seconds, func, /, *args, **kwargs):  
    time.sleep(seconds)  
    return func(*args, **kwargs)  
  
def duration(*, seconds, minutes, hours):  
    return seconds + 60 * minutes + 3600 * hours  
  
after(5, duration, seconds=20, minutes=3, hours=2)
```

В этом коде `seconds` передается как ключевой аргумент. Но он предназначен для применения с функцией `duration`, передаваемой `after()`. Использование только позиционных аргументов в `after()` предотвращает конфликт имен с аргументом `seconds`, который стоит на первом месте.

5.8. ИМЕНА, СТРОКИ ДОКУМЕНТАЦИИ И АННОТАЦИИ ТИПОВ

Стандартное соглашение об именах функций рекомендует использовать строчные буквы с разделением слов символами подчеркивания (`_`) — `read_data()`, а не `readData()`. Если функция не предназначена для прямого использования, так как она является вспомогательной или какой-либо деталью внутренней реализации, перед ее именем обычно ставится один символ подчеркивания — `_helper()`. Впрочем, это только соглашения. Вы можете назвать функцию как угодно, если это имя будет действительным идентификатором.

Имя функции можно получить с помощью атрибута `__name__`. Это может пригодиться при отладке.

```
>>> def square(x):
...     return x * x
...
>>> square.__name__
'square'
>>>
```

Первая строка функции обычно содержит строку документации с описанием использования этой функции:

```
def factorial(n):
    """
    Вычисляет факториал n. Пример:

    >>> factorial(6)
    120
    >>>
    """
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

Строка документации хранится в атрибуте `__doc__` функции. Часто IDE обращаются к ней для вывода интерактивной справки.

Функции могут снабжаться аннотациями типов:

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Аннотации типов не влияют на процесс вычисления функции. Другими словами, присутствие аннотаций никак не увеличивает скорость и не дает дополнительной проверки ошибок на стадии выполнения. Аннотации просто сохраняются в атрибуте `__annotations__` функции — словаре, связывающем имена аргументов с переданными аннотациями. Сторонние инструменты (IDE и средства проверки кода) могут использовать аннотации для разных целей.

Иногда встречаются аннотации типов, присоединенные к локальным переменным внутри функции:

```
def factorial(n:int) -> int:
    result: int = 1      # Локальная переменная с аннотацией типа
    while n > 1:
```

```
    result *= n
    n -= 1
return result
```

Такие аннотации полностью игнорируются интерпретатором. Они не проверяются, не сохраняются и даже не вычисляются. Еще раз: аннотации типов существуют только для предоставления информации сторонним средствам проверки кода. Добавлять аннотации типов к функциям не рекомендуется, если только вы не используете средства проверки типа, обрабатывающие эти аннотации. При задании аннотаций типов легко ошибиться. Если не найти ошибку специальными инструментами, она так и останется незамеченной, пока ваш код не обработает инструментами проверки типов кто-то другой.

5.9. ПРИМЕНЕНИЕ ФУНКЦИЙ И ПЕРЕДАЧА ПАРАМЕТРОВ

Параметры при вызове функции — локальные имена, которые связываются с передаваемыми входными объектами. Python передает предоставленные объекты в функцию как есть, без дополнительного копирования. При передаче изменяемых объектов (списков или словарей) нужна особая осторожность. Вносимые в них изменения отразятся на исходном объекте:

```
def square(items):
    for i, x in enumerate(items):
        items[i] = x * x          # Изменить items на месте

a = [1, 2, 3, 4, 5]
square(a) # Changes a to [1, 4, 9, 16, 25]
```

Говорят, что у функций, изменяющих свои входные значения или состояние других частей программы «за кулисами», есть побочные эффекты, которых лучше избегать. Они могут стать источником неочевидных ошибок программирования при увеличении размеров и сложности программ. По вызову функции может быть трудно определить наличие у нее побочных эффектов. Такие функции плохо взаимодействуют с программами, включающими потоки и параллелизм, так как побочные эффекты обычно должны защищаться блокировками.

Важно отличать изменение объекта от переназначения имени переменной. Возьмем следующую функцию:

```
def sum_squares(items):
    items = [x*x for x in items]    # Переназначить имя "items"
```

```

    return sum(items)

a = [1, 2, 3, 4, 5]
result = sum_squares(a)
print(a)                                # [1, 2, 3, 4, 5] (не изменяется)

```

Может показаться, что здесь функция `sum_squares()` перезаписывает переданную переменную `items`. Да, локальной метке `items` присваивается новое значение. Но начальное входное значение (`a`) не изменяется операцией. Вместо этого `items` связывается с другим объектом — результатом внутреннего спискового включения. Присваивание переменной и изменение объектов — не одно и то же. Присваивая значение имени переменной, вы не перезаписываете объект под этим именем — имя просто связывается с другим объектом.

Стилистически функции с побочными эффектами часто возвращают результат `None`. Рассмотрим метод `sort()` списка:

```

>>> items = [10, 3, 2, 9, 5]
>>> items.sort()                # Обратите внимание: нет возвращаемого
>>> items                        # значения
[2, 3, 5, 9, 10]
>>>

```

Метод `sort()` сортирует элементы списка «на месте». Он не возвращает результат. Отсутствие результата — веский признак побочного эффекта, в нашем случае — перестановки элементов списка.

Иногда уже существует последовательность или отображение с данными, которые должны передаваться функции. Для этого при вызове используются обозначения `*` и `**`:

```

def func(x, y, z):
    ...

s = (1, 2, 3)

# Передача последовательности как аргументов
result = func(*s)

# Передача отображения как ключевых аргументов
d = { 'x':1, 'y':2, 'z':3 }
result = func(**d)

```

Вы можете получать данные из нескольких источников и даже передавать часть аргументов явно. Все будет работать, если нет дублирования и функция получает обязательные аргументы, которые правильно сочетаются

с сигнатурой вызова. * и ** даже могут многократно использоваться в одном вызове функции. Если вы пропустите аргумент или укажете несколько значений для одного, то получите ошибку. Python никогда не позволит вызвать функцию с аргументами, не соответствующими его сигнатуре.

5.10. ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Команда `return` возвращает значение из функции. Если оно не задано или этой команды нет, возвращается `None`. Для возврата нескольких значений заключите их в кортеж:

```
def parse_value(text):
    """
    Разбивает текст в форме name=val на (name, val)
    """
    parts = text.split('=', 1)
    return (parts[0].strip(), parts[1].strip())
```

Значения, возвращаемые в кортеже, можно распаковать на отдельные переменные:

```
name, value = parse_value('url=http://www.python.org')
```

Иногда вместо этого используются именованные кортежи:

```
from typing import NamedTuple

class ParseResult(NamedTuple):
    name: str
    value: str

def parse_value(text):
    """
    Разбивает текст в форме name=val на (name, val)
    """
    parts = text.split('=', 1)
    return ParseResult(parts[0].strip(), parts[1].strip())
```

Именованный кортеж работает, как и обычный (с ним можно выполнять те же операции и распаковку), но к возвращаемым значениям можно обращаться по именам атрибутов:

```
r = parse_value('url=http://www.python.org')
print(r.name, r.value)
```

5.11. ОБРАБОТКА ОШИБОК

Одна из проблем с функцией `parse_value()` из прошлого раздела связана с обработкой ошибок. Что нужно предпринять, если входной текст сформирован некорректно и правильный результат вернуть нельзя?

Результат может рассматриваться как необязательный — то есть функция либо работает и возвращает ответ, либо возвращает `None`. Это обычно указывает на отсутствие значения. Функцию можно привести к следующему виду:

```
def parse_value(text):
    parts = text.split('=', 1)
    if len(parts) == 2:
        return ParseResult(parts[0].strip(), parts[1].strip())
    else:
        return None
```

При такой структуре бремя проверки необязательного результата возлагается на сторону вызова:

```
result = parse_value(text)
if result:
    name, value = result
```

Или в более компактной записи, поддерживаемой в Python 3.8+:

```
if result := parse_value(text):
    name, value = result
```

Вместо возвращения `None` можно рассматривать некорректно сформированный текст как ошибку и выдать исключение:

```
def parse_value(text):
    parts = text.split('=', 1)
    if len(parts) == 2:
        return ParseResult(parts[0].strip(), parts[1].strip())
    else:
        raise ValueError('Bad value')
```

В этом случае сторона вызова сможет обрабатывать неверные значения конструкцией `try-except`:

```
try:
    name, value = parse_value(text)
    ...
except ValueError:
    ...
```


Ответ на вопрос об использовании исключений не всегда однозначен. В общем случае исключения — более популярный способ обработки аномальных результатов. Но часто возникающие исключения обходятся дорого. Если вы пишете код, критичный по скорости, возвращение `None`, `False`, `-1` или другого специального значения как признака ошибки может оказаться предпочтительным.

5.12. ПРАВИЛА МАСШТАБИРОВАНИЯ

При каждом выполнении функции создается локальное пространство имен — среда с именами и значениями параметров функции, а также все переменные, значения которых присваиваются внутри тела функции. Привязка имен известна при определении функции, а все имена, присвоенные в теле функции, связываются с локальной средой. Все остальные имена, которые используются в теле функции без присваивания (свободные переменные), динамически ищутся в глобальном пространстве имен — это всегда модуль, в котором определяется функция.

Есть два типа ошибок, относящихся к именам, которые могут происходить при выполнении функции. Обращение к неопределенному имени свободной переменной в глобальной среде приводит к исключению `NameError`. Обращение к локальной переменной без присвоенного значения приводит к исключению `UnboundLocalError`. Последняя ошибка часто возникает из-за ошибок последовательности выполнения:

```
def func(x):
    if x > 0:
        y = 42
    return x + y # Значение y не присваивается, если условие ложно
```

```
func(10)    # Возвращает 52
func(-10)   # UnboundLocalError: обращение к y до присваивания
```

Исключение `UnboundLocalError` иногда возникает в случае неосторожного использования операторов присваивания «на месте». Команда `n += 1` обрабатывается по схеме `n = n + 1`. Если это происходит до присваивания исходного значения `n`, попытка завершится неудачей.

```
def func():
    n += 1      # Ошибка: UnboundLocalError
```

Важно, что имена переменных никогда не переходят в другую область видимости. Переменные бывают глобальными либо локальными, и это определяется

на момент определения функции. Следующий пример показывает, как это делается:

```
x = 42
def func():
    print(x) # Ошибка. UnboundLocalError
    x = 13

func()
```

Здесь может показаться, что функция `print()` будет выводить значение глобальной переменной `x`. Но последующее присваивание `x` помечает ее как локальную переменную. Ошибка возникает из-за обращения к локальной переменной, которой еще не было присвоено значение.

Если удалить функцию `print()`, вы получите код, выглядящий так, словно он заново присваивает значение глобальной переменной:

```
x = 42
def func():
    x = 13
func()
# x все еще содержит 42
```

При выполнении этого кода `x` сохраняет свое значение 42, но кажется, что код изменяет глобальную переменную `x` из функции `func`. Переменные внутри функции всегда рассматриваются как локальные. В результате переменная `x` в теле функции обозначает новый объект со значением 13, а не внешнюю переменную. Для изменения этого поведения используйте команду `global`. Она объявляет имена как принадлежащие к глобальному пространству имен и необходима, когда нужно изменить глобальную переменную. Пример:

```
x = 42
y = 37
def func():
    global x # 'x' в глобальном пространстве имен
    x = 13
    y = 0
func()
# x теперь содержит 13. y по-прежнему содержит 37.
```

Заметьте, что использование команды `global` — дурной тон для Python. Если вы пишете код, где функция должна «за кулисами» изменять состояние, лучше использовать определение класса с изменением состояния через изменение экземпляра или переменной класса. Пример:

```
class Config:
    x = 42

def func():
    Config.x = 13
```

В Python поддерживаются вложенные определения функций:

```
def countdown(start):
    n = start
    def display(): # Вложенное определение функции
        print('T-minus', n)
    while n > 0:
        display()
        n -= 1
```

Переменные во вложенных функциях связаны с использованием лексической области видимости. Иначе говоря, имена сначала ищутся в локальной области, а потом последовательно в охватывающих областях видимости от внутренней до внешней. Еще раз: процесс не динамический — связывание переменных определяется в точке определения функции на основании синтаксиса. Как и в случае с глобальными переменными, внутренние функции не могут присвоить новое значение локальной переменной, определенной во внешней функции. Например, следующий код работать не будет:

```
def countdown(start):
    n = start
    def display():
        print('T-minus', n)
    def decrement():
        n -= 1 # Ошибка: UnboundLocalError
    while n > 0:
        display()
        decrement()
```

Для решения проблемы объявите переменную `n` с ключевым словом `nonlocal`:

```
def countdown(start):
    n = start
    def display():
        print('T-minus', n)
    def decrement():
        nonlocal n
        n -= 1 # Изменяет внешнюю переменную n
    while n > 0:
        display()
        decrement()
```

`nonlocal` не может использоваться для ссылок на локальную переменную — это должна быть ссылка на локальную переменную во внешней области видимости. Поэтому, если функция присваивает значение глобальной переменной, вам все равно придется использовать объявление `global`.

Использование вложенных функций и объявлений `nonlocal` — тоже плохой тон в программировании. Например, у внутренних функций нет внешней видимости, что может усложнить тестирование и отладку. Но вложенные функции иногда бывают полезны для разбиения сложных вычислений на меньшие части и сокрытия подробностей внутренней реализации.

5.13. РЕКУРСИЯ

Python поддерживает рекурсивные функции:

```
def sumn(n):
    if n == 0:
        return 0
    else:
        return n + sumn(n-1)
```

Но для глубины рекурсивных вызовов функций установлено ограничение. Функция `sys.getrecursionlimit()` возвращает текущую максимальную глубину рекурсии, а `sys.setrecursionlimit()` может использоваться для его изменения. Значение по умолчанию равно **1000**. Его можно увеличить, но программы все еще ограничиваются размером стека, установленным операционной системой. При превышении максимальной глубины рекурсии выдается исключение `RuntimeError`. При увеличении лимита до слишком высокого уровня возможен фатальный сбой Python с ошибкой сегментации или другой ошибкой операционной системы.

На практике проблемы с ограничением рекурсии возникают только при работе с глубоко вложенными рекурсивными структурами данных, такими как деревья и графы. Для многих алгоритмов, работающих с деревьями, естественно, подходят рекурсивные решения, и, если ваша структура данных слишком велика, возможен выход за границу стека. Но есть некоторые обходные пути: пример приведен в главе 6 при рассмотрении генераторов.

5.14. ЛЯМБДА-ФУНКЦИИ

Анонимная (неименованная) функция может определяться лямбда-выражением:

`lambda` аргументы: выражение

Здесь аргументы — список аргументов, разделенных запятыми, а выражение — выражение, где эти аргументы используются:

```
a = lambda x, y: x + y
r = a(2, 3)      # r присваивается 5
```

Код, определяемый с ключевым словом `lambda`, должен быть действительным выражением. Лямбда-функции не могут занимать более одной строки, и в них не могут использоваться команды, не являющиеся выражениями (`try` и `while`). Лямбда-функции подчиняются тем же правилам масштабирования, что и обычные функции.

Одно из главных применений `lambda` — определение небольших функций обратного вызова. Например, лямбда-функции нередко встречаются при использовании со встроенными операциями, такими как `sorted()`:

```
# Сортировка списка слов по количеству уникальных букв
result = sorted(words, key=lambda word: len(set(word)))
```

Будьте осторожны, если лямбда-функция содержит свободные переменные (не указанные как параметры):

```
x = 2
f = lambda y: x * y
x = 3
g = lambda y: x * y
print(f(10))    # --> выводит 30
print(g(10))    # --> выводит 30
```

Здесь можно было бы ожидать, что вызов `f(10)` выведет `20`, отражая тот факт, что `x` был равен `2` во время определения. Но это не так.

Так как переменная свободная, при вычислении `f(10)` будет использоваться значение `x` на момент вычисления. Оно может отличаться от значения переменной на момент определения лямбда-функции. Иногда это поведение называется *поздним связыванием*.

Чтобы зафиксировать значение переменной на момент определения, используйте аргумент по умолчанию:

```
x = 2
f = lambda y, x=x: x * y
x = 3
g = lambda y, x=x: x * y
print(f(10))    # --> выводит 20
print(g(10))    # --> выводит 30
```

Этот способ работает, потому что значения аргументов по умолчанию вычисляются только в точке определения функции, поэтому отражают текущее значение `x`.

5.15. ФУНКЦИИ ВЫСШЕГО ПОРЯДКА

Python поддерживает концепцию *функций высшего порядка*, значит функции могут передаваться как аргументы другим функциям, помещаться в структуры данных и возвращаться функциями. Говорят, что функции — *первоклассные объекты*. Это значит, что работа с ними не отличается от работы с любым другим видом данных. Ниже приведен пример функции, получающей на входе другую функцию и вызывающей ее после небольшой задержки, например для эмуляции быстрогодействия микросервиса в облаке:

```
import time

def after(seconds, func):
    time.sleep(seconds)
    func()

# Example usage
def greeting():
    print('Hello World')

after(10, greeting) # Выводит 'Hello World' через 10 секунд
```

Здесь аргумент `func` функции `after()` — пример так называемой *функции обратного вызова* (callback). Название объясняется тем, что функция `after()` вызывает «в обратном направлении» функцию, переданную в аргументе.

Когда функция передается как данные, она неявно сохраняет информацию об окружении, где была определена. Допустим, `greeting()` использует переменную так:

```
def main():
    name = 'Guido'
    def greeting():
        print('Hello', name)
    after(10, greeting) # Выдает: 'Hello Guido'

main()
```

Здесь переменная `name` используется в `greeting()`. Но это локальная переменная внешней функции `main()`. Когда `greeting` передается `after()`, функция

запоминает свою среду и использует значение необходимой переменной `name`. При этом используется механизм, именуемый *замыканием* (closure). Замыкание состоит из функции и среды со всеми переменными для выполнения тела функции.

Замыкания и вложенные функции полезны при написании кода, основанного на концепции отложенного вычисления. `after()` — пример этой концепции. Она получает функцию, которая выполняется не немедленно, а в какой-то момент в будущем. Это распространенный паттерн программирования, который встречается и в других контекстах. Например, программа может содержать функции, которые выполняются при возникновении каких-то событий — нажатий клавиш, перемещений мыши, поступления сетевых пакетов и т. д. Во всех этих случаях выполнение функции откладывается до момента, когда происходит что-то важное. После выполнения функции замыкание обеспечит ее всей необходимой информацией.

Можно писать функции, которые создают и возвращают другие функции:

```
def make_greeting(name):
    def greeting():
        print('Hello', name)
    return greeting

f = make_greeting('Guido')
g = make_greeting('Ada')

f()      # Выводит: 'Hello Guido'
g()      # Выводит: 'Hello Ada'
```

Здесь `make_greeting()` не выполняет никаких содержательных вычислений. Она создает и возвращает функцию `greeting()`, которая делает реальную работу. Это происходит только при выполнении этой функции в будущем.

В этом примере две переменные `f` и `g` содержат две разные версии функции `greeting()`. И хотя `make_greeting()`, создавшая эти функции, уже не выполняется, `greeting()` все еще помнит переменную `name`, которая была определена. Она — часть замыкания каждой функции.

По поводу замыканий: связывание имен переменных — не «моментальный снимок», а динамический процесс. Замыкание указывает на переменную `name` и значение, которое было ей присвоено последним. Это тонкий момент, но следующий пример показывает, какие трудности могут возникнуть:

```
def make_greetings(names):
    funcs = []
    for name in names:
        funcs.append(lambda: print('Hello', name))
```

```

    return funcs

# Проверка
a, b, c = make_greetings(['Guido', 'Ada', 'Margaret'])
a()    # Выводит 'Hello Margaret'
b()    # Выводит 'Hello Margaret'
c()    # Выводит 'Hello Margaret'

```

Здесь создается список функций (с использованием `lambda`). Может показаться, что все они используют уникальное значение `name`, так как функция изменяется при каждой итерации цикла. Но это не так. Все функции в итоге используют одно и то же значение `name` — то, которое было у переменной при возврате из внешней функции `make_greetings()`.

Такое поведение будет неожиданным, и этого нам не нужно. Если вы хотите сохранить копию переменной, сохраните ее в аргументе по умолчанию, как было описано выше:

```

def make_greetings(names):
    funcs = []
    for name in names:
        funcs.append(lambda name=name: print('Hello', name))
    return funcs

# Проверка
a, b, c = make_greetings(['Guido', 'Ada', 'Margaret'])
a()    # Выводит 'Hello Guido'
b()    # Выводит 'Hello Ada'
c()    # Выводит 'Hello Margaret'

```

В двух последних примерах функции определялись с помощью `lambda`. Часто лямбда-функции используются как сокращенная запись для создания небольших функций обратного вызова. Но это необязательно. Фрагмент тоже можно переписать в следующем виде:

```

def make_greetings(names):
    funcs = []
    for name in names:
        def greeting(name=name):
            print('Hello', name)
        funcs.append(greeting)
    return funcs

```

Выбор места и времени использования лямбда-функции определяется личными предпочтениями и зависит от ясности кода. Если код становится труднее читать, этого следует избегать.

5.16. ПЕРЕДАЧА АРГУМЕНТОВ ФУНКЦИЯМ ОБРАТНОГО ВЫЗОВА

Одна серьезная проблема с функциями обратного вызова связана с передачей аргументов передаваемой функции. Возьмем написанную ранее функцию `after()`:

```
import time

def after(seconds, func):
    time.sleep(seconds)
    func()
```

В этом коде `func()` фиксируется для вызова без аргументов. Если вы захотите передать дополнительные аргументы, можно попытаться сделать так:

```
def add(x, y):
    print(f'{x} + {y} -> {x+y}')
    return x + y

after(10, add(2, 3)) # Ошибка: add() вызывается немедленно
```

Здесь функция `add(2, 3)` выполняется немедленно, возвращая 5. Потом `after()` вызывает ошибку, пытаясь выполнить `5()`. Это определенно не то, чего вы ожидали. Но на первый взгляд нет очевидного способа заставить программу работать при вызове `add()` с нужными аргументами.

Это указывает на более масштабную проблему проектирования, связанную с использованием функций и функциональным программированием вообще, — композицию функций. Когда функции по-разному сочетаются, нужно думать, как соединяются их входные и выходные данные. Это не всегда просто.

В нашем случае одно из возможных решений основано на упаковке вычисления в функцию с нулем аргументов при помощи `lambda`:

```
after(10, lambda: add(2, 3))
```

Такие маленькие функции с нулем аргументов иногда называют *преобразователями* (*thunk*). Это выражение, которое будет вычислено позднее, когда будет вызвано как функция с нулем аргументов. Этот способ может стать приемом общего назначения, позволяющим отложить вычисление любого выражения на будущее. Поместите выражение в `lambda` и вызовите функцию, когда вам понадобится значение.

Вместо использования лямбда-функции можно воспользоваться вызовом `functools.partial()` для создания частично вычисленной функции:

```
from functools import partial

after(10, partial(add, 2, 3))
```

`partial()` создает вызываемый объект, один или несколько аргументов которого уже были заданы и кешированы. Это может быть удобным способом привести неподходящие функции в соответствие с ожидаемыми сигнатурами в обратных вызовах и других возможных применениях. Несколько примеров использования `partial()`:

```
def func(a, b, c, d):
    print(a, b, c, d)

f = partial(func, 1, 2)      # Зафиксировать a=1, b=2
f(3, 4) # func(1, 2, 3, 4)
f(10, 20) # func(1, 2, 10, 20)

g = partial(func, 1, 2, d=4) # Зафиксировать a=1, b=2, d=4
g(3) # func(1, 2, 3, 4)
g(10) # func(1, 2, 10, 4)
```

`partial()` и `lambda` могут использоваться для похожих целей. Но между этими двумя решениями есть важные семантические различия. С `partial()` вычисление и связывание аргументов происходит при первом определении частичной функции. При использовании лямбда-функции с нулем аргументов вычисление и связывание аргументов выполняется во время фактического выполнения лямбда-функции (все вычисления откладываются):

```
>>> def func(x, y):
...     return x + y
...
>>> a = 2
>>> b = 3
>>> f = lambda: func(a, b)
>>> g = partial(func, a, b)
>>> a = 10
>>> b = 20
>>> f() # Использует текущие значения a, b
30
>>> g() # Использует текущие значения a, b
5
>>>
```

Частичные выражения вычисляются полностью, поэтому вызов `partial()` создает объекты, способные сериализоваться в последовательности байтов, сохраняться в файлах и даже передаваться по сети (например, средствами модуля стандартной библиотеки `pickle`). С лямбда-функциями это невозможно. Поэтому в приложениях с передачей функций (возможно, с интерпретаторами Python, работающими в разных процессах или на разных компьютерах) решение с `partial()` оказывается более гибким.

Замечу, что применение частичных функций тесно связано с концепцией, называемой *каррированием* (currying). Это прием функционального программирования, где функция с несколькими аргументами выражается цепочкой вложенных функций с одним аргументом:

```
# Функция с тремя аргументами
def f(x, y, z):
    return x + y + z

# Каррированная версия
def fc(x):
    return lambda y: (lambda z: x + y + z)

# Пример использования
a = f(2, 3, 4) # Функция с тремя аргументами
b = fc(2)(3)(4) # Каррированная версия
```

Этот прием не относится к общепринятому стилю программирования Python, и причины для его практического применения встречаются редко. Но иногда слово «каррирование» мелькает в разговорах с программистами, которые провели много времени, разбираясь в таких вещах, как лямбда-числение. Этот метод обработки нескольких аргументов был назван в честь знаменитого логика Хаскелла Карри. Полезно знать, что это такое, например, если вы столкнетесь с группой функциональных программистов, ожесточенно спорящих на каком-нибудь светском мероприятии.

Вернемся к исходной проблеме передачи аргументов. Другой вариант передачи аргументов функции обратного вызова основан на их передаче в отдельных аргументах внешней вызывающей функции. Рассмотрим следующую версию функции `after()`:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    func(*args)

after(10, add, 2, 3) # Вызывает add(2, 3) через 10 секунд
```

Заметьте, что передача ключевых аргументов `func()` не поддерживается. Это было сделано намеренно. Одна из проблем ключевых аргументов в том, что имена аргументов заданной функции могут конфликтовать с уже используемыми именами (то есть `seconds` и `func`). Ключевые аргументы могут быть зарезервированы для передачи параметров самой функции `after()`:

```
def after(seconds, func, *args, debug=False):
    time.sleep(seconds)
    if debug:
        print('About to call', func, args)
    func(*args)
```

Но не все потеряно. Задать ключевые аргументы для `func()` можно при помощи `partial()`:

```
after(10, partial(add, y=3), 2)
```

Если вы хотите, чтобы функция `after()` получала ключевые аргументы, безопасным решением может стать использование только позиционных аргументов:

```
def after(seconds, func, debug=False, /, *args, **kwargs):
    time.sleep(seconds)
    if debug:
        print('About to call', func, args, kwargs)
    func(*args, **kwargs)
```

```
after(10, add, 2, y=3)
```

Есть и другой настораживающий факт: `after()` представляет два разных вызова функций, объединенных вместе. Возможно, проблема передачи аргументов может быть решена декомпозицией на две функции:

```
def after(seconds, func, debug=False):
    def call(*args, **kwargs):
        time.sleep(seconds)
        if debug:
            print('About to call', func, args, kwargs)
        func(*args, **kwargs)
    return call
```

```
after(10, add)(2, y=3)
```

Теперь конфликты между аргументами `after()` и `func` полностью исключены. Но такие решения могут породить конфликты с вашими коллегами, которые будут читать ваш код.

5.17. ВОЗВРАЩЕНИЕ РЕЗУЛЬТАТОВ ИЗ ОБРАТНЫХ ВЫЗОВОВ

В прошлом разделе не упоминалась еще одна проблема: возвращение результатов вычислений. Рассмотрим измененную функцию `after()`:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    return func(*args)
```

Она работает, но есть неочевидные граничные случаи, возникающие из-за того, что в ней задействованы две разные функции: сама функция `after()` и переданный обратный вызов `func`.

Одна из сложностей связана с обработкой исключений. Попробуйте следующие два примера:

```
after("1", add, 2, 3) # Ошибка: TypeError (ожидается целое число)
after(1, add, "2", 3) # Ошибка: TypeError (конкатенация int
                      # со str невозможна)
```

В обоих случаях выдается ошибка `TypeError`, но по разным причинам и в разных функциях. Первая ошибка обусловлена проблемой в самой функции `after()`: функции `time.sleep()` передается неправильный аргумент. Вторая ошибка возникает из-за проблемы с выполнением функции обратного вызова `func(*args)`.

Есть несколько вариантов, чтобы различить эти два случая. В одном из них используются цепочки исключений. Идея в том, чтобы упаковать ошибки из обратного вызова особым способом, позволяющим обрабатывать их отдельно от других ошибок:

```
class CallbackError(Exception):
    pass

def after(seconds, func, *args):
    time.sleep(seconds)
    try:
        return func(*args)
    except Exception as err:
        raise CallbackError('Callback function failed') from err
```

Измененный код отделяет ошибки от переданного обратного вызова в отдельную категорию исключений. Он используется примерно так:

```
try:
    r = after(delay, add, x, y)
```

```
except CallbackError as err:
    print("It failed. Reason", err.__cause__)
```

При возникновении проблемы с выполнением самой функции `after()` это исключение будет распространяться наружу без перехвата. С другой стороны, проблемы, связанные с выполнением переданной функции обратного вызова, будут перехватываться, и программа будет сообщать о них исключением `CallbackError`.

Вся эта схема неочевидна. Но на практике обработка ошибок — достаточно сложная тема. Такой подход позволяет точнее управлять распределением ответственности и упрощает документирование поведения `after()`. Если с обратным вызовом возникают проблемы, программа всегда сообщает о ней в виде `CallbackError`.

Другой вариант — упаковка результата функции обратного вызова в экземпляр-результат, содержащий и значение, и ошибку. Например, класс можно определить так:

```
class Result:
    def __init__(self, value=None, exc=None):
        self._value = value
        self._exc = exc
    def result(self):
        if self._exc:
            raise self._exc
        else:
            return self._value
```

Далее используйте этот класс для возвращения результатов из функции `after()`:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    try:
        return Result(value=func(*args))
    except Exception as err:
        return Result(exc=err)
```

Пример использования:

```
r = after(1, add, 2, 3)
print(r.result())           # Выводит 5
s = after("1", add, 2, 3)   # Немедленно выдает TypeError -
                           # недопустимый аргумент sleep().
t = after(1, add, "2", 3)   # Возвращает "Result"
print(t.result())           # Выдает TypeError
```

Второй способ основан на выделении выдачи результата функции обратного вызова в отдельный шаг. При возникновении проблемы с `after()` о ней будет сообщено немедленно. Если возникнет проблема с обратным вызовом `func()`, уведомление о ней будет отправлено при попытке пользователя получить результат вызовом метода `result()`.

Этот стиль упаковки результата в специальный экземпляр для распаковки в будущем все чаще встречается в современных языках программирования. Одна из причин в том, что он упрощает проверку типов. Если вам понадобится включить аннотацию типа в `after()`, ее поведение полностью определено — она всегда возвращает `Result` и ничего другого:

```
def after(seconds, func, *args) -> Result:
    ...
```

Этот паттерн еще не так часто встречается в коде Python, но он регулярно возникает при работе с примитивами синхронизации (потокками и процессами). Например, экземпляры `Future` ведут себя так при работе с пулами потоков:

```
from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor(16)
r = pool.submit(add, 2, 3) # Возвращает Future
print(r.result())         # Распаковывает результат Future
```

5.18. ДЕКОРАТОРЫ

Декоратор — это функция, создающая обертку для другой функции. Ее главная цель — изменение или расширение поведения упакованного объекта. Декораторы обозначаются специальным символом `@`:

```
@decorate
def func(x):
    ...
```

Приведенный код — сокращенная запись следующего фрагмента:

```
def func(x):
    ...
func = decorate(func)
```

Здесь определяется функция `func()`. Но сразу же после ее определения сам объект функции передается функции `decorate()`, которая возвращает объект, заменяющий исходную версию `func`.

В качестве примера конкретной реализации можно привести декоратор `@trace`, добавляющий отладочные сообщения к функциям:

```
def trace(func):
    def call(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return call

# Пример использования
@trace
def square(x):
    return x * x
```

В этом коде `trace()` создает функцию-обертку. Она выводит отладочную информацию и вызывает исходный объект функции. Так, при вызове `square()` вы увидите вывод функции `print()` в обертке.

Но не все так просто! На практике функции содержат и метаданные: имя функции, строку документации, аннотации типов и т. д. При заключении функции в обертку эта информация скрывается. Во время написания декоратора используйте `@wraps()`:

```
from functools import wraps

def trace(func):
    @wraps(func)
    def call(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return call
```

Декоратор `@wraps()` копирует разные метаданные функции в заменяющую функцию. В нашем случае метаданные из функции `func()` копируются в возвращаемую функцию-обертку `call()`.

Декораторы должны быть в отдельной строке перед функцией. К функции можно применить несколько декораторов. Пример:

```
@decorator1
@decorator2
def func(x):
    pass
```

В этом случае декораторы применяются так:

```
def func(x):
    pass

func = decorator1(decorator2(func))
```


Порядок применения декораторов важен. Например, в определениях классов такие декораторы, как `@classmethod` и `@staticmethod`, часто должны размещаться на внешнем уровне:

```
class SomeClass(object):
    @classmethod    # Да
    @trace
    def a(cls):
        pass

    @trace          # Нет. Ошибка
    @classmethod
    def b(cls):
        pass
```

Причина такого ограничения связана со значениями, возвращаемыми `@classmethod`. Иногда декоратор возвращает объект, отличный от обычной функции. Если сам внешний декоратор этого не ожидает, может произойти сбой. В этом случае `@classmethod` создает объект `classmethod` (см. главу 7). Если только декоратор не был написан с учетом этого факта, перечисление декораторов в неправильном порядке приведет к ошибке.

Декоратор может получать аргументы. Допустим, вы хотите изменить декоратор `@trace`, чтобы он поддерживал нестандартное сообщение:

```
@trace("You called {func.__name__}")
def func():
    pass
```

При передаче аргументов семантика процесса декорирования выглядит так:

```
def func():
    pass

# Создать функцию-декоратор
temp = trace("You called {func.__name__}")

# Применить его к func
func = temp(func)
```

Здесь внешняя функция, получающая аргумент, отвечает за создание функции-декоратора. Затем она вызывается с декорируемой функцией для получения результата. Реализация декоратора выглядит так:

```
from functools import wraps

def trace(message):
    def decorate(func):
```

```

    @wraps(func)
    def wrapper(*args, **kwargs):
        print(message.format(func=func))
        return func(*args, **kwargs)
    return wrapper
return decorate

```

У этой реализации есть одна интересная особенность: внешняя функция — это своего рода фабрика декораторов. Допустим, вам приходится писать такой код:

```

@trace('You called {func.__name__}')
def func1():
    pass

@trace('You called {func.__name__}')
def func2():
    pass

```

Это однообразие быстро утомляет. Код можно упростить: вызовите внешнюю функцию-декоратор один раз и повторно используйте результат:

```

logged = trace('You called {func.__name__}')

@logged
def func1():
    pass

@logged
def func2():
    pass

```

Декораторы не обязаны заменять исходную функцию. Иногда они просто выполняют какое-то действие — скажем, регистрацию. Например, при построении реестра обработчиков событий можно определить декоратор, который работает так:

```

@eventhandler('BUTTON')
def handle_button(msg):
    ...

@eventhandler('RESET')
def handle_reset(msg):
    ...

```

Декоратор, управляющий регистрацией, выглядит так:

```

# Декоратор обработчиков событий
_event_handlers = { }
def eventhandler(event):

```

```
def register_function(func):
    _event_handlers[event] = func
    return func
return register_function
```

5.19. ОТОБРАЖЕНИЕ, ФИЛЬТРАЦИЯ И СВЕРТКА

Программисты, знакомые с функциональными языками, часто интересуются стандартными операциями списков — отображением, фильтрацией и сверткой. Большая часть этой функциональности обеспечивается списковыми включениями и генераторными выражениями:

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
squares = [ square(x) for x in nums ] # [1, 4, 9, 16, 25]
```

Формально даже эта короткая однострочная функция не нужна. Можно использовать следующую запись:

```
squares = [ x * x for x in nums ]
```

Фильтрация тоже может выполняться списковым включением:

```
a = [ x for x in nums if x > 2 ] # [3, 4, 5]
```

При использовании выражения-генератора вы получите генератор, который производит результаты поэтапно с помощью перебора:

```
squares = (x*x for x in nums)      # Создает генератор
for n in squares:
    print(n)
```

Python предоставляет встроенную функцию `map()`, эквивалентную отображению функции с выражением-генератором. Пример выше можно записать так:

```
squares = map(lambda x: x*x, nums)
for n in squares:
    print(n)
```

Встроенная функция `filter()` создает генератор, фильтрующий значения:

```
for n in filter(lambda x: x > 2, nums):
    print(n)
```

Для накопления или свертки значений используется функция `functools.reduce()`:

```
from functools import reduce
total = reduce(lambda x, y: x + y, nums)
```

В обобщенной форме `reduce()` получает функцию с двумя аргументами, итерируемый объект и исходное значение. Несколько примеров:

```
nums = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, nums)          # 15
product = reduce(lambda x, y: x * y, nums, 1)     # 120

pairs = reduce(lambda x, y: (x, y), nums, None)
# (((((None, 1), 2), 3), 4), 5)
```

`reduce()` накапливает значения слева направо для переданного итерируемого объекта. Ниже приведен псевдокод функции `reduce(func, items, initial)`:

```
def reduce(func, items, initial):
    result = initial
    for item in items:
        result = func(result, item)
    return result
```

Применение `reduce()` часто приводит к недоразумениям. Более того, некоторые стандартные операции свертки (`sum()`, `min()` и `max()`) уже есть во встроенном виде. Ваш код станет понятнее (и, скорее всего, будет быстрее выполняться), если вы воспользуетесь одной из этих функций, а не будете пытаться реализовать стандартные операции на базе `reduce()`.

5.20. ИНТРОСПЕКЦИЯ, АТТРИБУТЫ И СИГНАТУРЫ

Мы уже знаем, что функции — это объекты. Значит, их можно присваивать переменным, помещать в структуры данных и использовать так же, как и любую разновидность данных в программе. Их можно по-разному анализировать. В табл. 5.1 приведены некоторые часто используемые атрибуты функций. Многие из них пригодятся при отладке, ведении журналов и других операциях с функциями.

Атрибут `f.__name__` содержит имя, которое использовалось при определении функции. `f.__qualname__` — более длинное имя с дополнительной информацией о среде.

Таблица 5.1. Атрибуты функций

| АТТРИБУТ | ОПИСАНИЕ |
|--------------------------------|---|
| <code>f.__name__</code> | Имя функции |
| <code>f.__qualname__</code> | Полное имя функции (для вложенных) |
| <code>f.__module__</code> | Имя модуля, где определена функция |
| <code>f.__doc__</code> | Строка документации |
| <code>f.__annotations__</code> | Аннотации типов |
| <code>f.__globals__</code> | Словарь с глобальным пространством имен |
| <code>f.__closure__</code> | Переменные замыкания (если есть) |
| <code>f.__code__</code> | Нижележащий объект с кодом |

Атрибут `f.__module__` содержит строку с именем модуля, в котором определена функция. Атрибут `f.__globals__` содержит словарь, который служит глобальным пространством имен для функции. Обычно это тот же словарь, который присоединяется к соответствующему объекту модуля.

`f.__doc__` содержит строку документации функции. Атрибут `f.__annotations__` содержит словарь с аннотациями типов, если они есть.

`f.__closure__` хранит ссылки на значения переменных замыкания для вложенных функций. Эти переменные глубоко спрятаны, но следующий пример показывает, как их просматривать:

```
def add(x, y):
    def do_add():
        return x + y
    return do_add

>>> a = add(2, 3)
>>> a.__closure__
(<cell at 0x10edf1e20: int object at 0x10ecc1950>,
 <cell at 0x10edf1d90: int object at 0x10ecc1970>)
>>> a.__closure__[0].cell_contents
2
>>>
```

Объект `f.__code__` отображает откомпилированный байт-код тела функции.

К функциям можно присоединять и произвольные атрибуты:

```
def func():
    команды

func.secure = 1
func.private = 1
```

Атрибуты не видны из тела функции — это не локальные переменные, и их нет в среде выполнения в виде имен. Атрибуты функций предназначены для хранения дополнительных метаданных. Некоторые фреймворки или разные средства метапрограммирования используют пометку функций, то есть присоединение атрибутов к функциям. Один из примеров — декоратор `@abstractmethod`. Он используется с методами внутри абстрактных базовых классов. Декоратор не делает ничего, кроме присоединения атрибута:

```
def abstractmethod(func):
    func.__isabstractmethod__ = True
    return func
```

Другой блок кода (в нашем случае — метакласс) проверяет атрибут и использует его для дополнительных проверок при создании экземпляра.

Чтобы получить больше информации о параметрах функции для получения ее сигнатуры, можно воспользоваться `inspect.signature()`:

```
import inspect

def func(x: int, y:float, debug=False) -> float:
    pass

sig = inspect.signature(func)
```

Объекты сигнатур дают удобные средства вывода и получения подробной информации о параметрах:

```
# Вывод сигнатуры в удобочитаемой форме
print(sig) # Выводит (x: int, y: float, debug=False) -> float

# Получение списка имен аргументов
print(list(sig.parameters)) # Выводит [ 'x', 'y', 'debug' ]

# Перебор параметров и вывод различных метаданных
for p in sig.parameters.values():
    print('name', p.name)
    print('annotation', p.annotation)
    print('kind', p.kind)
    print('default', p.default)
```

Сигнатура — это метаданные, описывающие природу функции: как она вызывается, аннотации типов и т. д. С ней можно выполнить ряд операций. Одна из них — сравнение. Вот как можно узнать, имеют ли две функции одинаковые сигнатуры:

```
def func1(x, y):
    pass

def func2(x, y):
    pass

assert inspect.signature(func1) == inspect.signature(func2)
```

Подобные сравнения могут быть полезны во фреймворках. Например, фреймворк может использовать сравнение сигнатур для проверки соответствия написанных функций или методов ожидаемому прототипу.

Сигнатура в атрибуте `__signature__` функции будет выводиться в справочных сообщениях и возвращаться при вызове `inspect.signature()`:

```
def func(x, y, z=None):
    ...

func.__signature__ = inspect.signature(lambda x,y: None)
```

Здесь необязательный аргумент `z` будет скрываться при проверке `func`. Вызов `inspect.signature()` будет возвращать присоединенную сигнатуру.

5.21. АНАЛИЗ СРЕДЫ

Функции могут проверять свою исполнительную среду с помощью встроенных функций `globals()` и `locals()`. `globals()` возвращает словарь с глобальным пространством имен. Он совпадает с содержимым атрибута `func.__globals__`. Обычно это тот же словарь, где хранится содержимое внешнего модуля. `locals()` возвращает словарь со значениями всех локальных переменных и переменных замыкания. Этот словарь — не реальная структура данных, где хранятся эти переменные. Локальные переменные могут происходить из внешних функций (через замыкание) или определяться внутри. `locals()` собирает все эти переменные и помещает в словарь за вас. Изменение элемента в словаре `locals()` не влияет на соответствующую переменную:

```
def func():
    y = 20
    locs = locals()
    locs['y'] = 30 # Попытаемся изменить y
    print(locs['y']) # Выводит 30
    print(y)        # Выводит 20
```

Чтобы изменение вступило в силу, вам придется скопировать его обратно в локальную переменную обычным присваиванием:

```
def func():
    y = 20
    locs = locals()
    locs['y'] = 30
    y = locs['y']
```

Функция может получить свой стековый кадр функцией `inspect.currentframe()`. Функция может получить стековый кадр своей вызывающей стороны, отслеживая трассировку стека через атрибуты `f.f_back` кадра:

```
import inspect

def spam(x, y):
    z = x + y
    grok(z)

def grok(a):
    b = a * 10

    # Выводит: {'a':5, 'b':50 }
    print(inspect.currentframe().f_locals)

    # Выводит: {'x':2, 'y':3, 'z':5 }
    print(inspect.currentframe().f_back.f_locals)

spam(2, 3)
```

Иногда для получения стековых кадров можно воспользоваться функцией `sys._getframe()`:

```
import sys
def grok(a):
    b = a * 10
    print(sys._getframe(0).f_locals) # Свой кадр
    print(sys._getframe(1).f_locals) # Сторона вызова
```

Для анализа кадров могут использоваться атрибуты из табл. 5.2.

Таблица 5.2. Атрибуты кадра стека

| АТРИБУТ | ОПИСАНИЕ |
|-----------------------|--|
| <code>f.f_back</code> | Предыдущий стековый кадр (по направлению к стороне вызова) |
| <code>f.f_code</code> | Выполняемый объект с кодом |

| АТРИБУТ | ОПИСАНИЕ |
|---------------------------|--|
| <code>f.f_locals</code> | Словарь локальных переменных (<code>locals()</code>) |
| <code>f.f_globals</code> | Словарь, используемый для глобальных переменных (<code>globals()</code>) |
| <code>f.f_builtins</code> | Словарь, используемый для встроенных имен |
| <code>f.f_lineno</code> | Номер строки |
| <code>f.f_lasti</code> | Текущая инструкция – индекс в строке байт-кода из <code>f_code</code> |
| <code>f.f_trace</code> | Функция, вызываемая в начале каждой строки исходного кода |

Просмотр стековых кадров полезен при отладке и анализе кода. Ниже приведена интересная отладочная функция для просмотра значений переменных на стороне вызова:

```
import inspect
from collections import ChainMap

def debug(*varnames):
    f = inspect.currentframe().f_back
    vars = ChainMap(f.f_locals, f.f_globals)
    print(f'{f.f_code.co_filename}:{f.f_lineno}')
    for name in varnames:
        print(f' {name} = {vars[name]!r}')

# Пример использования
def func(x, y):
    z = x + y
    debug('x', 'y') # Выводит x и y с указанием файла/строки
    return z
```

5.22. ДИНАМИЧЕСКОЕ ВЫПОЛНЕНИЕ И СОЗДАНИЕ КОДА

Функция `exec(str [, globals [, locals]])` выполняет строку с произвольным кодом Python. Код, передаваемый `exec()`, выполняется так, как если бы он находился на месте операции `exec`:

```
a = [3, 5, 10, 13]
exec('for i in a: print(i)')
```

Передаваемый `exec()` код выполняется в локальном и глобальном пространствах имен стороны вызова. Но учтите, что изменения в локальных переменных неэффективны:

```
def func():
    x = 10
    exec("x = 20")
    print(x)           # Выводит 10
```

Это объясняется тем, что `locals` — словарь с собранными локальными переменными, а не они сами (подробности см. в следующем разделе).

`exec()` может получать один-два объекта словарей, которые содержат глобальное и локальное пространство имен для выполняемого кода:

```
globs = {'x': 7,
         'y': 10,
         'birds': ['Parrot', 'Swallow', 'Albatross']}
locs = { }

# Выполнение с использованием приведенных выше словарей
# как глобального и локального пространств имен
exec('z = 3 * x + 4 * y', globs, locs)
exec('for b in birds: print(b)', globs, locs)
```

Если опустить одно или оба пространства имен, используются текущие значения глобального и/или локального пространств. Если предоставить словарь только для `globals`, он используется как для глобального, так и локального пространства.

Динамическое выполнение кода часто используется для создания функций и методов. Следующая функция создает для класса метод `__init__()` с заданным списком имен:

```
def make_init(*names):
    parms = ','.join(names)
    code = f'def __init__(self, {parms}):\n'
    for name in names:
        code += f' self.{name} = {name}\n'
    d = { }
    exec(code, d)
    return d['__init__']

# Пример использования
class Vector:
    __init__ = make_init('x','y','z')
```

Этот прием используется во многих частях стандартной библиотеки. Например, `namedtuple()`, `@dataclass` и другие аспекты зависят от динамического создания кода вызовом `exec()`.

5.23. АСИНХРОННЫЕ ФУНКЦИИ И AWAIT

Python поддерживает ряд языковых средств, связанных с асинхронным выполнением кода. К их числу относятся *асинхронные функции* и *ожидаемые объекты* (awaitable). Они используются в основном в программах, где задействован параллелизм и модуль `asyncio`. Но другие библиотеки тоже могут пользоваться ими.

Для определения асинхронной функции поставьте дополнительное ключевое слово `async` перед нормальным определением функции:

```
async def greeting(name):
    print(f'Hello {name}')
```

Вы обнаружите, что функция не выполняется обычным способом. Собственно, она вообще не выполняется. Вместо этого вы получаете объект асинхронной функции:

```
>>> greeting('Guido')
<coroutine object greeting at 0x104176dc8>
>>>
```

Чтобы функция заработала, она должна выполняться под контролем другого кода. Чаще всего для этого используется модуль `asyncio`:

```
>>> import asyncio
>>> asyncio.run(greeting('Guido'))
Hello Guido
>>>
```

Здесь мы видим важнейшую особенность асинхронных функций: они никогда не выполняются сами по себе. Для этого всегда нужен некий менеджер или библиотечный код. Им не обязан быть модуль `asyncio`, как в этом примере. Но всегда есть сторона, обеспечивающая выполнение асинхронных функций.

Не считая управляемого выполнения, асинхронная функция работает точно так же, как и любая другая функция Python. Команды выполняются по порядку, работают все стандартные средства управления последовательностью выполнения.

Чтобы вернуть результат, используйте стандартную команду `return`:

```
async def make_greeting(name):
    return f'Hello {name}'
```

Значение, переданное `return`, возвращается внешней функцией `run()`, которая использовалась для выполнения асинхронной функции:

```
>>> import asyncio
>>> a = asyncio.run(make_greeting('Paula'))
>>> a
'Hello Paula'
>>>
```

Асинхронные функции могут вызывать другие асинхронные функции с помощью `await`-выражений:

```
async def make_greeting(name):
    return f'Hello {name}'

async def main():
    for name in ['Paula', 'Thomas', 'Lewis']:
        a = await make_greeting(name)
        print(a)

# Выполнить функцию. Выводятся приветствия для Paula, Thomas и Lewis
asyncio.run(main())
```

Использовать `await` можно только в границах внешнего определения асинхронной функции. Оно является и обязательной частью выполнения асинхронных функций. Опустив `await`, вы увидите, что код работать не будет.

Требование использования `await` намекает на основную проблему с использованием асинхронных функций. Другая модель вычислений не позволяет использовать их в сочетании с другими частями Python. А конкретно — невозможно написать код, вызывающий асинхронную функцию из неасинхронной:

```
async def twice(x):
    return 2 * x

def main():
    print(twice(2))          # Ошибка. Функция не выполняется.
    print(await twice(2))   # Ошибка. Здесь не может использоваться await.
```

Объединение асинхронной и неасинхронной функциональности в том же приложении — сложная тема. Особенно если учитывать некоторые приемы программирования с участием функций высшего порядка, обратных вызовов и декораторов. В большинстве случаев поддержка асинхронных функций должна строиться как особый случай.

Именно так поступает Python с протоколами итератора и менеджера контекста. Например асинхронный менеджер контекста может быть определен методами `__aenter__()` и `__aexit__()`:

```
class AsyncManager(object):
    def __init__(self, x):
        self.x = x

    async def yow(self):
        pass

    async def __aenter__(self):
        return self

    async def __aexit__(self, ty, val, tb):
        pass
```

Заметьте, что эти методы — асинхронные функции. Поэтому они могут выполнять другие асинхронные функции с использованием `await`. Для использования такого менеджера нужно применить специальный синтаксис `async with`, который допустим только внутри асинхронной функции:

```
# Пример использования
async def main():
    async with AsyncManager(42) as m:
        await m.yow()

asyncio.run(main())
```

Класс может также найти асинхронный итератор, определяя методы `__aiter__()` и `__anext__()`. Они используются командой `async for`, которая может находиться только в асинхронных функциях.

С практической точки зрения асинхронная функция ведет себя точно так же, как обычная. Просто она должна выполняться в управляемой среде (такой как `asyncio`). Если вы не решили сознательно работать в такой среде, лучше обходить асинхронные функции стороной. Вы избавите себя от множества проблем.

5.24. НАПОСЛЕДОК: О ФУНКЦИЯХ И КОМПОЗИЦИИ

Любая система строится как композиция компонентов. В Python к числу таких компонентов относятся разные библиотеки и объекты. Но в основе

всего лежат функции. Это «клей», связывающий части системы, и основной механизм передачи данных.

По большей части эта глава рассказывает о природе функций и их интерфейсах. Как входные данные предоставляются функции? Как обрабатываются? Как выводятся сообщения об ошибках? Как сделать все эти аспекты более контролируемыми и понятными?

Работая над крупным проектом, стоит задуматься о взаимодействиях функций как возможном источнике проблем. Часто от этого зависит, что вы получите в результате — интуитивно понятный, удобный API или бесформенную мешанину.

ГЛАВА 6

Генераторы

Функции-генераторы — одна из самых интересных и мощных возможностей Python. Генераторы часто описываются как удобный способ определения новых разновидностей паттернов перебора. Но этим их полезность не ограничивается. Они могут на фундаментальном уровне изменять всю модель выполнения функций. В этой главе рассматриваются генераторы, их стандартные применения, сопрограммы на их базе и делегирование.

6.1. ГЕНЕРАТОРЫ И YIELD

Если функция использует ключевое слово `yield`, она определяет объект, называемый генератором. Цель применения генератора — производство значений, используемых в переборе:

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1
```

```
# Пример использования
for x in countdown(10):
    print('T-minus', x)
```

При вызове этой функции вы обнаружите, что ее код не выполняется:

```
>>> c = countdown(10)
>>> c
<generator object countdown at 0x105f73740>
>>>
```

Вместо этого создается объект-генератор. Генератор выполняет функцию только тогда, когда вы начнете перебор по нему. В одном из способов для него вызывается функция `next()`:

```
>>> next(c)
Counting down from 10
10
>>> next(c)
9
```

При вызове `next()` функция-генератор выполняет команды до достижения `yield`. Команда `yield` возвращает результат, после чего выполнение функции приостанавливается до следующего вызова `next()`. Во время приостановки функция сохраняет все локальные переменные и среду выполнения. При возобновлении выполнение продолжается с команды, следующей за `yield`.

`next()` — сокращение вызова метода `__next__()` для генератора. Можно поступить так:

```
>>> c.__next__()
8
>>> c.__next__()
7
>>>
```

Обычно вы не вызываете `next()` для генератора напрямую, а используете команду `for` или другую операцию, потребляющую элементы:

```
for n in countdown(10):
    команды

a = sum(countdown(10))
```

Функция-генератор производит элементы, пока не вернет управление из-за достижения конца функции или выполнения команды `return`. При этом выдается исключение `StopIteration`, завершающее цикл `for`. Если функция-генератор возвращает значение, отличное от `None`, оно присоединяется к исключению `StopIteration`. Следующая функция-генератор использует как `yield`, так и `return`:

```
def func():
    yield 37
    return 42
```

А вот так этот код будет выполняться:

```
>>> f = func()
>>> f
<generator object func at 0x10b7cd480>
```



```
>>> next(f)
37
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 42
>>>
```

Заметьте, что возвращаемое значение присоединяется к `StopIteration`. Для его получения нужно явно перехватить `StopIteration` и извлечь значение:

```
try:
    next(f)
except StopIteration as e:
    value = e.value
```

Обычно функции-генераторы не возвращают значения. Генераторы почти всегда потребляются циклами `for`. Это не позволяет получить значение исключения. Значит, единственный разумный способ получить значение основан на ручном продвижении генератора явными вызовами `next()`. В большей части кода, где используются генераторы, этого не делается.

С генераторами связан еще один нюанс, проявляющийся при частичном потреблении функции-генератора. Рассмотрим следующий код с преждевременным выходом из цикла:

```
for n in countdown(10):
    if n == 2:
        break
команды
```

Здесь цикл `for` прерывается вызовом `break` и связанный с ним генератор никогда не отработывает до завершения. Чтобы функция-генератор выполнила некоторые завершающие действия, используйте `try-finally` или менеджер контекста:

```
def countdown(n):
    print('Counting down from', n)
    try:
        while n > 0:
            yield n
            n = n - 1
    finally:
        print('Only made it to', n)
```

Генераторы гарантированно выполняют код блока `finally`, даже если они не были потреблены полностью. Блок будет выполнен, когда прерванный

генератор уничтожится сборщиком мусора. Точно так же гарантируется выполнение любого завершающего кода с участием менеджера контекста:

```
def func(filename):
    with open(filename) as file:
        ...
        yield data
        ...
    # Здесь файл будет закрыт, даже если генератор был прерван
```

Корректное освобождение ресурсов — непростая задача. Но пока вы используете такие конструкции, как `try-finally` или менеджер контекста, генераторы отработают правильно даже в случае преждевременного прерывания.

6.2. ПЕРЕЗАПУСКАЕМЫЕ ГЕНЕРАТОРЫ

Обычно функция-генератор выполняется только один раз:

```
>>> c = countdown(3)
>>> for n in c:
...     print('T-minus', n)
...
T-minus 3
T-minus 2
T-minus 1
>>> for n in c:
...     print('T-minus', n)
...
>>>
```

Чтобы создать объект, допускающий повторные итерации, определите его как класс и сделайте метод `__iter__()` генератором:

```
class countdown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1
```

Такое решение работает, так как при каждом переборе метод `__iter__()` создает новый генератор.

6.3. ДЕЛЕГИРОВАНИЕ

Неотъемлемое свойство генераторов в том, что функция с `yield` никогда не выполняется сама по себе. Ею всегда должен управлять другой код с циклом `for` или явными вызовами `next()`. Это усложняет написание библиотечных функций с `yield`, ведь вызова функции-генератора недостаточно для обеспечения ее выполнения. Решить эту проблему можно с помощью команды `yield from`:

```
def countup(stop):
    n = 1
    while n <= stop:
        yield n
        n += 1

def countdown(start):
    n = start
    while n > 0:
        yield n
        n -= 1

def up_and_down(n):
    yield from countup(n)
    yield from countdown(n)
```

`yield from` делегирует процесс перебора внешней конструкции. Можно написать следующий код для управления перебором:

```
>>> for x in up_and_down(5):
...     print(x, end=' ')
1 2 3 4 5 5 4 3 2 1
>>>
```

`yield from` избавляет вас от необходимости управлять перебором вручную. Иначе вам пришлось бы записать `up_and_down(n)` так:

```
def up_and_down(n):
    for x in countup(n):
        yield x
    for x in countdown(n):
        yield x
```

Конструкция `yield from` особенно полезна при написании кода, который должен рекурсивно перебирать вложенные итерируемые объекты. Следующий код сглаживает вложенные списки:

```
def flatten(items):
    for i in items:
        if isinstance(i, list):
            yield from flatten(i)
        else:
            yield i
```

Пример использования:

```
>>> a = [1, 2, [3, [4, 5], 6, 7], 8]
>>> for x in flatten(a):
...     print(x, end=' ')
...
1 2 3 4 5 6 7 8
>>>
```

Одно из ограничений такой реализации связано с тем, что на нее распространяются ограничения рекурсии Python и она не сможет обработать структуры с глубокой вложенностью. Эту тему мы рассмотрим в следующем разделе.

6.4. ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ ГЕНЕРАТОРОВ

На первый взгляд неочевидно, как использовать генераторы для решения практических задач, кроме определения простых итераторов. Но генераторы оказываются особенно эффективны для структурирования разных задач обработки данных, связанных с конвейерами и рабочими процессами.

Генераторы могут стать полезным инструментом реструктуризации кода, состоящего из глубоко вложенных циклов `for` и условных конструкций. Рассмотрим сценарий, который ищет в каталоге с файлами Python все комментарии со словом `spam`:

```
import pathlib
import re

for path in pathlib.Path('.').rglob('*.py'):
    if path.exists():
        with path.open('rt', encoding='latin-1') as file:
            for line in file:
                m = re.match('.*(#.*)$', line)
                if m:
                    comment = m.group(1)
                    if 'spam' in comment:
                        print(comment)
```

Обратите внимание на глубину вложенности встроенных конструкций. Даже от одного взгляда на этот код становится не по себе. А теперь взгляните на следующую версию с использованием генераторов:

```
import pathlib
import re

def get_paths(topdir, pattern):
    for path in pathlib.Path(topdir).rglob(pattern):
        if path.exists():
            yield path

def get_files(paths):
    for path in paths:
        with path.open('rt', encoding='latin-1') as file:
            yield file

def get_lines(files):
    for file in files:
        yield from file

def get_comments(lines):
    for line in lines:
        m = re.match('.*(#.*)$', line)
        if m:
            yield m.group(1)

def print_matching(lines, substring):
    for line in lines:
        if substring in line:
            print(substring)

paths = get_paths('.', '*.py')
files = get_files(paths)
lines = get_lines(files)
comments = get_comments(lines)
print_matching(comments, 'spam')
```

В этом разделе задача делится на меньшие автономные компоненты. Каждый из них относится к конкретной задаче. Например, генератор `get_paths()` связан только с путями, `get_files()` — только с открытием файлов и т. д. Лишь в конце эти генераторы связываются воедино для решения задачи.

Создание небольших изолированных компонентов — хороший прием абстракции. Для примера возьмем `get_comments()`. На входе он получает любой итерируемый объект, производящий строки текста. Этот текст может поступать почти из любого источника — из файла, списка, генератора и т. д.

Эта функциональность оказывается намного более мощной и адаптируемой, чем если бы была встроена в серию вложенных циклов `for` с файлами. Так, благодаря генераторам код можно использовать повторно за счет разбиения задач на небольшие и четко определенные вычислительные. Меньшие задачи более удобны и для анализа, отладки и тестирования.

Генераторы полезны для изменения обычных правил вычисления при применении функций. Обычно применяемая функция выполняется немедленно и выдает результат. Генераторы этого не делают. При применении функции-генератора ее выполнение откладывается, пока другой код не вызовет для нее `next()` (явно или в цикле `for`).

Рассмотрим следующую функцию-генератор для сглаживания вложенных списков:

```
def flatten(items):
    for i in items:
        if isinstance(i, list):
            yield from flatten(i)
        else:
            yield i
```

Один из недостатков этой реализации в том, что из-за ограничения рекурсии в Python программа не будет работать со структурами глубокой вложенности. Проблему можно решить, управляя перебором с использованием стека:

```
def flatten(items):
    stack = [ iter(items) ]
    while stack:
        try:
            item = next(stack[-1])
            if isinstance(item, list):
                stack.append(iter(item))
            else:
                yield item
        except StopIteration:
            stack.pop()
```

Эта реализация строит внутренний стек итераторов. Она не подвержена ограничению рекурсии Python, потому что помещает данные во внутренний список, а не создает фреймы во внутреннем стеке интерпретатора. Так, если вам нужно свести несколько миллионов слоев какой-то необычайно глубокой структуры данных, вы обнаружите, что эта версия отлично работает.

Означают ли эти примеры, что весь ваш код нужно переписать с применением экзотических паттернов генераторов? Нет. Суть в том, что отложенное

вычисление генераторов позволяет вам изменить пространственно-временные измерения нормального вычисления функции. Есть разные практические сценарии, где эти приемы могут быть полезны и применяются неожиданным образом.

6.5. РАСШИРЕННЫЕ ГЕНЕРАТОРЫ И ВЫРАЖЕНИЯ **YIELD**

Внутри функции-генератора команда **yield** может использоваться как выражение в правой части оператора присваивания:

```
def receiver():
    print('Ready to receive')
    while True:
        n = yield
        print('Got', n)
```

Функция, так использующая **yield**, иногда называется расширенным генератором или сопрограммой на базе генератора. Но эти термины немного неточны и вводят в заблуждение, ведь сопрограмма чаще связывается с асинхронными функциями. Для предотвращения путаницы мы будем использовать термин «расширенный генератор», чтобы было понятно, что речь идет о стандартных функциях, использующих **yield**.

Функция, использующая **yield** как выражение, остается генератором, но используется иначе. Вместо того чтобы производить значения, она выполняется в ответ на значения, которые ей передаются:

```
>>> r = receiver()
>>> r.send(None)      # Продвигается к первому yield
Ready to receive
>>> r.send(1)
Got 1
>>> r.send(2)
Got 2
>>> r.send('Hello')
Got Hello
>>>
```

Здесь исходный вызов **r.send(None)** нужен, чтобы генератор выполнил команды, ведущие к первому выражению **yield**. В этот момент он приостанавливается, ожидая отправки ему значения методом **send()** связанного объекта-генератора **r**. Значение, переданное **send()**, возвращается выражением

`yield` в генераторе. При получении значения генератор выполняет команды до обнаружения следующего `yield`.

В приведенном виде функция выполняется бесконечно. Для закрытия генератора можно воспользоваться методом `close()`:

```
>>> r.close()
>>> r.send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Операция `close()` выдает исключение `GeneratorExit` внутри генератора в текущей команде `yield`. Это приводит к незаметному завершению генератора. При желании можно перехватить его для выполнения завершающих действий. Если после закрытия генератору будут передаваться дополнительные значения, выдается исключение `StopIteration`.

Исключения могут выдаваться внутри генератора методом `throw(ty [,val [,tb]])`, где `ty` — тип исключения, `val` — аргумент исключения (или кортеж аргументов), а `tb` — необязательная трассировка:

```
>>> r = receiver()
Ready to receive
>>> r.throw(RuntimeError, "Dead")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "receiver.py", line 14, in receiver
    n = yield
RuntimeError: Dead
>>>
```

Выдаваемые так исключения распространяются от текущей выполняемой команды `yield` в генераторе. Он может решить перехватить исключение и обработать его по своему усмотрению. Если генератор не обработает исключение, оно распространяется из него для обработки на более высоком уровне.

6.6. ПРИМЕНЕНИЕ РАСШИРЕННЫХ ГЕНЕРАТОРОВ

Расширенные генераторы — странная программная конструкция. В отличие от простого генератора, который естественным образом предоставляет данные

для цикла `for`, в расширенном генераторе нет функции основного языка. Тогда зачем писать функцию, которой должны отправляться значения?

Традиционно расширенные генераторы использовались в библиотеках, базирующихся на асинхронном вводе/выводе. В этом контексте они обычно называются сопрограммами или сопрограммами на базе генераторов. Но большая часть этой функциональности была интегрирована в функциональности `async` и `await` языка Python. Причин для использования `yield` в этом конкретном сценарии немного. Но практические применения все же есть.

Как и обычные генераторы, расширенный может использоваться для разных видов вычислений и управления последовательностью выполнения. Одним из примеров служит декоратор `@contextmanager` из модуля `contextlib`:

```
from contextlib import contextmanager

@contextmanager
def manager():
    print("Entering")
    try:
        yield 'somevalue'
    except Exception as e:
        print("An error occurred", e)
    finally:
        print("Leaving")
```

Здесь генератор используется для объединения двух половин менеджера контекста. Вспомните, что они определяются объектами, реализующими следующий протокол:

```
class Manager:
    def __enter__(self):
        return somevalue

    def __exit__(self, ty, val, tb):
        if ty:
            # Произошло исключение
            ...
            # Вернуть True, если исключение обработано,
            # или False в противном случае
```

С декоратором `@contextmanager` все, что предшествует команде `yield`, выполняется при входе в менеджер (с использованием метода `__enter__()`). Все после `yield` выполняется при выходе из него (с использованием метода `__exit__()`). В случае ошибки информация о ней передается в виде исключения в команде `yield`:

```

>>> with manager() as val:
...     print(val)
...
Entering
somevalue
Leaving
>>> with manager() as val:
...     print(int(val))
...
Entering
An error occurred invalid literal for int() with base 10: 'somevalue'
Leaving
>>>

```

Для реализации этой функциональности используется класс-обертка. Следующая упрощенная реализация показывает основную идею:

```

class Manager:
    def __init__(self, gen):
        self.gen = gen

    def __enter__(self):
        # Выполнение до yield
        return self.gen.send(None)

    def __exit__(self, ty, val, tb):
        # Распространить исключение (если оно есть)
        try:
            if ty:
                try:
                    self.gen.throw(ty, val, tb)
                except ty:
                    return False
            else:
                self.gen.send(None)
        except StopIteration:
            return True

```

Другое применение расширенных генераторов — использование функций для инкапсуляции «рабочей» задачи. Одна из важнейших особенностей вызова функции в том, что она формирует среду локальных переменных. Доступ к ним высоко оптимизирован. Он работает намного быстрее обращения к атрибутам классов и экземплярам.

Генератор продолжает существовать до явного закрытия или уничтожения. Поэтому можно воспользоваться им для создания задачи с долгим сроком жизни. Пример генератора, который получает байтовые фрагменты и собирает из них строки:

```
def line_receiver():
    data = bytearray()
    line = None
    linecount = 0
    while True:
        part = yield line
        linecount += part.count(b'\n')
        data.extend(part)
        if linecount > 0:
            index = data.index(b'\n')
            line = bytes(data[:index+1])
            data = data[index+1:]
            linecount -= 1
        else:
            line = None
```

Здесь генератор получает байтовые фрагменты, которые собираются в байтовый массив. Если массив содержит символ новой строки, строка извлекается и возвращается. В противном случае возвращается None. Следующий пример показывает, как это работает:

```
>>> r = line_receiver()
>>> r.send(None)          # Инициализация генератора
>>> r.send(b'hello')
>>> r.send(b'world\nit ')
b'hello world\n'
>>> r.send(b'works!')
>>> r.send(b'\n')
b'it works!\n'
>>>
```

Такой код может быть записан в виде класса:

```
class LineReceiver:
    def __init__(self):
        self.data = bytearray()
        self.linecount = 0
    def send(self, part):
        self.linecount += part.count(b'\n')
        self.data.extend(part)
        if self.linecount > 0:
            index = self.data.index(b'\n')
            line = bytes(self.data[:index+1])
            self.data = self.data[index+1:]
            self.linecount -= 1
            return line
        else:
            return None
```

Хотя написание класса может быть более привычным, код сложнее и работает медленнее. В ходе тестирования на компьютере автора передача большого набора блоков происходила на 40–50 % быстрее передачи генератора с этим кодом класса. Большая часть ускорения объясняется отказом от обращений к атрибутам экземпляров — локальные переменные работают быстрее.

Конечно, есть и другие возможные применения. Но важно помнить, что если вы встречаете использование `yield` в контексте, не связанном с перебором, скорее всего, это связано с использованием `send()` или `throw()`.

6.7. ГЕНЕРАТОРЫ И ИХ СВЯЗЬ С `AWAIT`

Классическое применение функций-генераторов — в библиотеках, связанных с асинхронным вводом/выводом (например, стандартный модуль `asyncio`). Но после выхода Python 3.5 большая часть этой функциональности переместилась в другие языковые средства, связанные с асинхронными функциями, и команду `await` (см. раздел 5.23).

`await` использует скрытое взаимодействие с генератором. Далее показан базовый протокол, используемый `await`:

```
class Awaitable:
    def __await__(self):
        print('About to await')
        yield # Должна быть генератором
        print('Resuming')

# Функция, совместимая с "await". Возвращает "awaitable"
def function():
    return Awaitable()

async def main():
    await function()
```

А вот как можно опробовать этот код с использованием `asyncio`:

```
>>> import asyncio
>>> asyncio.run(main())
About to await
Resuming
>>>
```

Так ли важно знать, как это работает? Вероятно, нет. Все эти механизмы обычно скрыты от вас. Но, работая с асинхронными функциями, помните,

что где-то внутри спрятана функция-генератор. Вы сможете ее найти, если достаточно глубоко зареетесь в решение технических проблем.

6.8. НАПОСЛЕДОК: КРАТКАЯ ИСТОРИЯ И ВОЗМОЖНОСТИ ГЕНЕРАТОРОВ

Генераторы — одна из самых интересных историй успеха в языке Python. Они являются частью более масштабной истории, связанной с перебором. Перебор — одна из самых распространенных операций в программировании. В ранних версиях Python перебор происходил с помощью индексирования последовательностей и метода `__getitem__()`. Последний преобразовался в текущий протокол перебора, основанный на методах `__iter__()` и `__next__()`.

Вслед за этим появились генераторы как более удобный механизм реализации итератора. В современном Python почти не осталось причин для самостоятельной итерации на базе чего-либо, кроме генератора. Даже для итерируемых объектов, которые вы можете реализовать сами, метод `__iter__()` удобно реализуется этим способом.

В более поздних версиях Python генераторы взяли на себя новую роль, поскольку они развили расширенные функции, связанные с сопрограммами: методы `send()` и `throw()`. Они уже не ограничивались перебором, но открывали возможности для использования генераторов в других контекстах. Это заложило основу для многих так называемых асинхронных фреймворков, используемых в сетевом и параллельном программировании.

Но с развитием асинхронного программирования большая часть этой функциональности преобразовалась в новые средства, использующие синтаксис `async/await`. По этой причине использование функций-генераторов за пределами контекста перебора (их исходного предназначения) встречается редко. Если вы самостоятельно определяете функцию-генератор, но не выполняете перебор, пересмотрите свой подход. Скорее всего, есть более эффективное или современное решение вашей задачи.

ГЛАВА 7

Классы и объектно-ориентированное программирование

Классы используются для создания новых разновидностей объектов. В этой главе подробно рассматриваются классы, но она не была задумана как справочник по объектно-ориентированному программированию и проектированию. Здесь описаны некоторые паттерны программирования, часто встречающиеся в Python, и способы, которыми вы можете настроить классы, чтобы они вели себя необычно.

Общая структура этой главы нисходящая: сначала в ней описываются высокоуровневые концепции и средства использования классов. Ближе к концу главы материал становится более техническим и сосредоточен на внутренней реализации.

7.1. ОБЪЕКТЫ

Почти весь код Python подразумевает создание объектов и выполнение операций с ними:

```
>>> s = "Hello World"
>>> s.upper()
'HELLO WORLD'
>>> s.replace('Hello', 'Hello Cruel')
'Hello Cruel World'
>>> s.split()
['Hello', 'World']
>>>
```

Пример использования объекта списка:

```
>>> names = ['Paula', 'Thomas']
>>> names.append('Lewis')
>>> names
['Paula', 'Thomas', 'Lewis']
>>> names[1] = 'Tom'
>>>
```

Важнейшая особенность любого объекта в том, что он обычно обладает некоторым состоянием (это могут быть символы строки, элементы списка и т. д.) и содержит методы, работающие с ним. Методы вызываются через сам объект, словно они — функции, присоединенные к нему оператором «точка» (.).

Любой объект связан с некоторым типом. Его можно узнать с помощью функции `type()`:

```
>>> type(names)
<class 'list'>
>>>
```

Объект называется экземпляром своего типа. Например `names` — экземпляр класса `list`.

7.2. КОМАНДА CLASS

Новые разновидности объектов определяются командой `class`. Класс обычно состоит из набора функций, формирующих его методы:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def inquiry(self):
        return self.balance
```

Важно, что команда `class` сама по себе не создает никаких экземпляров класса. Класс лишь содержит методы, которые станут доступны для экземпляров, созданных позднее. Это план или «чертеж» для изготовления объектов.

Функции, определяемые внутри класса, называются методами. Метод экземпляра — это функция, которая работает с экземпляром класса, переданным в первом аргументе. По общепринятым соглашениям ему присваивается имя `self`. В примере выше `deposit()`, `withdraw()` и `inquiry()` — методы экземпляров.

Методы `__init__()` и `__repr__()` — примеры специальных «волшебных» (или служебных) методов. Они имеют особое значение для среды выполнения интерпретатора. `__init__()` используется для инициализации состояния при создании нового экземпляра. `__repr__()` возвращает строку с представлением объекта. Определять этот метод не обязательно, но он упрощает процесс отладки и просмотр объектов из интерактивной оболочки.

Определение класса может включать строку документации и аннотации типов:

```
class Account:
    """
    Простой банковский счет
    """
    owner: str
    balance: float

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def inquiry(self):
        return self.balance
```

Аннотации типов никак не влияют на работу класса. Они не создают никаких дополнительных проверок. Это лишь метаданные, которые могут использоваться сторонними инструментами или IDE либо некоторыми программными средствами высокого уровня. В большинстве следующих примеров они использоваться не будут.

7.3. ЭКЗЕМПЛЯРЫ

Экземпляры класса создаются вызовом объекта класса как функции. При этом создается новый экземпляр, который затем передается методу `__init__()`. Аргументы `__init__()` состоят из только что созданного экземпляра `self` из аргументов, переданных при вызове объекта класса:

```
# Создание нескольких счетов

a = Account('Guido', 1000.0)
# Вызывает Account.__init__(a, 'Guido', 1000.0)

b = Account('Eva', 10.0)
# Вызывает Account.__init__(b, 'Eva', 10.0)
```

Внутри `__init__()` атрибуты сохраняются в экземпляре присваиванием `self`. Например, команда `self.owner = owner` сохраняет атрибут экземпляра. После возвращения вновь созданного экземпляра для обращения к этим атрибутам (и методам класса) можно использовать оператор «точка» (`.`):

```
a.deposit(100.0) # Вызывает Account.deposit(a, 100.0)
b.withdraw(50.00) # Вызывает Account.withdraw(b, 50.0)
owner = a.owner # Получить владельца счета
```

Важно, что у каждого экземпляра свое состояние. Для просмотра переменных экземпляров используется функция `vars()`:

```
>>> a = Account('Guido', 1000.0)
>>> b = Account('Eva', 10.0)
>>> vars(a)
{'owner': 'Guido', 'balance': 1000.0}
>>> vars(b)
{'owner': 'Eva', 'balance': 10.0}
>>>
```

Обратите внимание, что методы в список не включены. Они существуют на уровне класса. Каждый экземпляр содержит ссылку на свой класс, представленный его типом:

```
>>> type(a)
<class 'Account'>
>>> type(b)
<class 'Account'>
>>> type(a).deposit
<function Account.deposit at 0x10a032158>
>>> type(a).inquiry
<function Account.inquiry at 0x10a032268>
>>>
```

В следующем разделе подробнее рассматривается реализация связывания атрибутов и отношений между экземплярами и классами.

7.4. ОБРАЩЕНИЕ К АТТРИБУТАМ

С атрибутами экземпляров выполняются три основные операции: чтение, запись и удаление:

```
>>> a = Account('Guido', 1000.0)
>>> a.owner                # Чтение
'Guido'
>>> a.balance = 750.0      # Запись
>>> del a.balance          # Удаление
>>> a.balance
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Account' object has no attribute 'balance'
>>>
```

Почти все аспекты Python — динамические процессы с минимальными ограничениями. Ничто не мешает вам добавить новый атрибут к объекту после его создания:

```
>>> a = Account('Guido', 1000.0)
>>> a.creation_date = '2019-02-14'
>>> a.nickname = 'Former BDFL'
>>> a.creation_date
'2019-02-14'
>>>
```

Вместо выполнения этих операций оператором «точка» (.) можно передать имя атрибута в строковом виде функциям `getattr()`, `setattr()` и `delattr()`. Функция `hasattr()` проверяет наличие атрибута:

```
>>> a = Account('Guido', 1000.0)
>>> getattr(a, 'owner')
'Guido'
>>> setattr(a, 'balance', 750.0)
>>> delattr(a, 'balance')
>>> hasattr(a, 'balance')
False
>>> getattr(a, 'withdraw')(100)    # Вызов метода
>>> a
Account('Guido', 650.0)
>>>
```

Варианты `a.attr` и `getattr(a, 'attr')` взаимозаменяемы. Значит, запись `getattr(a, 'withdraw')(100)` аналогична `a.withdraw(100)`, несмотря на то что `withdraw()` — метод.

Заметьте, что функция `getattr()` получает необязательное значение по умолчанию. Обратиться к атрибуту, в существовании которого не уверены, можно так:

```
>>> a = Account('Guido', 1000.0)
>>> getattr(s, 'balance', 'unknown')
1000.0
>>> getattr(s, 'creation_date', 'unknown')
'unknown'
>>>
```

Обращаясь к методу как к атрибуту, вы получаете объект, называемый связанным методом:

```
>>> a = Account('Guido', 1000.0)
>>> w = a.withdraw
>>> w
<bound method Account.withdraw of Account('Guido', 1000.0)>
>>> w(100)
>>> a
Account('Guido', 900.0)
>>>
```

Связанный метод — это объект, содержащий как экземпляр (`self`), так и функцию, реализующую метод. Когда вы вызываете связанный метод, добавляя круглые скобки и аргументы, он выполняет метод, передавая присоединенный экземпляр в качестве первого аргумента. Например, вызов `w(100)` преобразуется в `Account.withdraw(a, 100)`.

7.5. ПРАВИЛА МАСШТАБИРОВАНИЯ

Хотя классы определяют изолированное пространство имен для методов, оно не служит областью для разрешения имен, используемых внутри методов. Поэтому при реализации класса ссылки на атрибуты и методы должны быть полностью уточнены. Например, в методах во всех ссылках на атрибуты экземпляра должно использоваться имя `self` — отсюда `self.balance`, а не `balance`. Это правило применяется и при вызове методов из другого метода. Допустим, вы хотите реализовать `withdraw()` в контексте внесения на счет отрицательных сумм:

```

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.deposit(-amount) # Необходимо использовать self.deposit()

    def inquiry(self):
        return self.balance

```

Отсутствие видимости уровня классов — одна из областей, где Python отличается от C++ и Java. Если вы использовали эти языки, параметр `self` в Python эквивалентен указателю `this`, не считая того, что в Python он должен использоваться явно.

7.6. ПЕРЕГРУЗКА ОПЕРАТОРОВ И ПРОТОКОЛЫ

В главе 4 рассматривалась модель данных Python. Особое внимание уделялось специальным методам, реализующим операторы и протоколы Python. Например, функция `len(obj)` вызывает `obj.__len__()`, а `obj[n]` — `obj.__getitem__(n)`.

При определении новых классов обычно определяют некоторые из этих методов. Одним из них был `__repr__()` в классе `Account`, предназначенный для улучшения отладочного вывода. Если вы создаете что-то посложнее (например, нестандартный контейнер), можно определить и другие специальные методы. Допустим, вы хотите создать класс, представляющий портфель банковских счетов:

```

class AccountPortfolio:
    def __init__(self):
        self.accounts = []

    def add_account(self, account):
        self.accounts.append(account)

    def total_funds(self):
        return sum(account.inquiry() for account in self)

    def __len__(self):
        return len(self.accounts)

```

```

def __getitem__(self, index):
    return self.accounts[index]

def __iter__(self):
    return iter(self.accounts)

# Пример
port = AccountPortfolio()
port.add_account(Account('Guido', 1000.0))
port.add_account(Account('Eva', 50.0))

print(port.total_funds()) # -> 1050.0
len(port) # -> 2

# Вывод счетов
for account in port:
    print(account)

# Обращение к отдельному счету по индексу
port[1].inquiry() # -> 50.0

```

Специальные методы в конце (`__len__()`, `__getitem__()` и `__iter__()`) обеспечивают работу `AccountPortfolio` с операторами Python, такими как индексирование и перебор.

Иногда в разговорах программистов встречается определение «питонический», например питонический код. Это неформальный термин, но обычно он означает, в какой степени объект соблюдает правила среды Python. Он подразумевает поддержку (разумную) базовых возможностей Python (перебор, индексирование и другие операции). Почти всегда это делается реализацией заранее определенных специальных методов (см. главу 4).

7.7. НАСЛЕДОВАНИЕ

Наследование — механизм создания нового класса. Он специализирует или изменяет поведение существующего класса. Исходный класс называется базовым, суперклассом или родительским классом. Новый класс называется производным, дочерним классом, субклассом или подтипом. Если класс создается наследованием, он наследует атрибуты, определенные его базовыми классами. Но производный класс может переопределить любые из них и добавить свои новые.

Наследование определяется списком имен базовых классов, разделенных запятыми, в команде `class`. Если базовый класс не указан, класс неявно наследуется от `object`. `object` — корневой класс для всех объектов Python. Он предоставляет реализацию по умолчанию для таких часто используемых методов, как `__str__()` и `__repr__()`.

Одно из возможных применений наследования — расширение существующего класса новыми методами. Допустим, вы хотите добавить в класс `Account` метод `panic()`, снимающий все средства со счета. Вот как это делается:

```
class MyAccount(Account):
    def panic(self):
        self.withdraw(self.balance)

# Example
a = MyAccount('Guido', 1000.0)
a.withdraw(23.0)                # a.balance = 977.0
a.panic()                      # a.balance = 0
```

Наследование может использоваться и для переопределения уже существующих методов. Ниже приведена специализированная версия `Account`. Она переопределяет метод `inquiry()` для периодического завышения баланса, чтобы невнимательный пользователь превысил кредитный лимит и заплатил большой штраф при внесении платежа по субстандартной ипотеке:

```
import random

class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return self.balance * 1.10
        else:
            return self.balance

a = EvilAccount('Guido', 1000.0)
a.deposit(10.0)                # Вызывает Account.deposit(a, 10.0)
available = a.inquiry()        # Вызывает EvilAccount.inquiry(a)
```

В этом примере экземпляры `EvilAccount` идентичны экземплярам `Account` во всем, кроме переопределения метода `inquiry()`.

Иногда производный класс реализует метод заново, но при этом он должен вызвать исходную реализацию. Для явного вызова исходного метода можно воспользоваться функцией `super()`:

```
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return 1.10 * super().inquiry()
        else:
            return super().inquiry()
```

В этом примере вызов `super()` позволяет обратиться к методу в виде, в котором он был определен ранее. Вызов `super().inquiry()` использует

исходное определение `inquiry()`, предшествовавшее его переопределению в `EvilAccount`.

Наследование может использоваться для добавления дополнительных атрибутов в экземпляры. Но этот вариант встречается реже. Следующий пример показывает, как ввести множитель 1.10 из прошлого примера в атрибут уровня экземпляра, который можно изменять на программном уровне:

```
class EvilAccount(Account):
    def __init__(self, owner, balance, factor):
        super().__init__(owner, balance)
        self.factor = factor

    def inquiry(self):
        if random.randint(0,4) == 1:
            return self.factor * super().inquiry()
        else:
            return super().inquiry()
```

Одна из неочевидных проблем при добавлении атрибутов связана с тем, что делать с существующим методом `__init__()`. В этом примере мы определяем новую версию `__init__()`, включающую дополнительную переменную экземпляра `factor`. Но при переопределении `__init__()` ответственность за инициализацию родителя возлагается на потомка. Для этого используется вызов `super().__init__()`, как показано выше. Если вы забудете сделать это, то у вас появится наполовину инициализированный объект и программа сломается. Так как инициализация родительского объекта требует дополнительных аргументов, они должны передаваться дочернему методу `__init__()`.

Наследование может приводить к появлению неочевидных ошибок. Взгляните на метод `__repr__()` класса `Account`:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'
```

Этот метод определяется, чтобы упростить процесс отладки за счет получения удобного вывода. Но он жестко запрограммирован для использования имени `Account`. При использовании наследования внезапно выяснится, что вывод содержит неправильную информацию:

```
>>> class EvilAccount(Account):
...     pass
... 
```

```
>>> a = EvilAccount('Eva', 10.0)
>>> a
Account('Eva', 10.0) # Обратите внимание на неправильный вывод
>>> type(a)
<class 'EvilAccount'>
>>>
```

Для исправления этого недостатка нужно изменить метод `__repr__()`, чтобы в нем использовалось правильное имя типа:

```
class Account:
    ...
    def __repr__(self):
        return f'{type(self).__name__}({self.owner!r}, {self.balance!r})'
```

Теперь мы видим более точный вывод. Наследование используется не со всеми классами. Но если вы планируете этот сценарий использования для класса, который пишете, обращайте внимание на подобные детали. В общем случае нужно избегать жесткого закрепления имен классов в коде.

Наследование устанавливает в системе типов особые отношения, где каждый дочерний класс успешно проходит проверку типа на принадлежность родительскому:

```
>>> a = EvilAccount('Eva', 10)
>>> type(a)
<class 'EvilAccount'>
>>> isinstance(a, Account)
True
>>>
```

Это «особое отношение»: `EvilAccount` — частный случай `Account`. Иногда это отношение наследования используется для определения онтологий или таксономий типов объектов:

```
class Food:
    pass

class Sandwich(Food):
    pass

class RoastBeef(Sandwich):
    pass

class GrilledCheese(Sandwich):
    pass

class Taco(Food):
    pass
```


На практике такая организация объектов может быть проблемной и рискованной. Допустим, вы захотите добавить в приведенную выше иерархию класс `HotDog`. Где он должен находиться? С одной стороны, кажется логичным сделать его субклассом `Sandwich`. С другой стороны, кто-то решит, что этот класс нужно сделать субклассом `Taco`. А может, вы решите его сделать субклассом обоих родительских классов:

```
class HotDog(Sandwich, Taco):  
    pass
```

В этот момент голова идет кругом и начинаются ожесточенные споры. Возможно, сейчас подходящий момент упомянуть о том, что Python поддерживает множественное наследование (то есть указание нескольких классов в качестве родительских). Полученный дочерний класс наследует объединенную функциональность всех родителей. За дополнительной информацией обращайтесь к разделу 7.19.

7.8. ОТКАЗ ОТ НАСЛЕДОВАНИЯ В ПОЛЬЗУ КОМПОЗИЦИИ

Одна из проблем, связанных с наследованием, — это наследование реализации. Допустим, вы хотите создать структуру данных стека с операциями занесения и извлечения элементов. Одно из быстрых решений — наследование от списка с добавлением нового метода:

```
class Stack(list):  
    def push(self, item):  
        self.append(item)
```

```
# Пример  
s = Stack()  
s.push(1)  
s.push(2)  
s.push(3)  
s.pop()    # -> 3  
s.pop()    # -> 2
```

Конечно, такая структура данных работает как стек. Но она и поддерживает все остальные возможности списков: вставку, сортировку, сегментное присваивание и т. д. Это называется наследованием реализации.

Наследование применяется для повторного использования кода, на основе которого строится что-то другое. Но заодно вы получаете множество функций,

никак не относящихся к решаемой задаче. Этот объект может показаться пользователю странным. Почему стек содержит методы сортировки?

Более правильным решением здесь будет композиция. Вместо того чтобы строить стек наследованием от списка, лучше построить стек как независимый класс со внутренним списком. Наличие внутреннего списка — это деталь реализации. Пример:

```
class Stack:
    def __init__(self):
        self._items = list()

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __len__(self):
        return len(self._items)
```

Пример использования

```
s = Stack()
s.push(1)
s.push(2)
s.push(3)
s.pop()      # -> 3
s.pop()      # -> 2
```

Этот объект работает как и прежде, но теперь он сосредоточен на функциональности стека. В нем нет никаких лишних методов списков или возможностей, не присущих стеку. Его предназначение становится более очевидным.

Небольшое расширение этой реализации может получать внутренний класс `list` в необязательном аргументе:

```
class Stack:
    def __init__(self, *, container=None):
        if container is None:
            container = list()
        self._items = container

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()
```

```
def __len__(self):  
    return len(self._items)
```

Преимущество такого подхода в том, что он поощряет слабое связывание компонентов. Можно создать стек, который хранит свои элементы в типизованном массиве вместо списка. Вот как это может выглядеть:

```
import array  
  
s = Stack(container=array.array('i'))  
s.push(42)  
s.push(23)  
s.push('a lot')    # TypeError
```

Это пример *внедрения зависимостей*. Вместо жесткой привязки `Stack` к `list` вы можете сделать его зависимым от любого контейнера, который пользователь решит передать, если он реализует требуемый интерфейс.

В более широком смысле преобразование внутреннего списка в подробность реализации связано с проблемой абстракции данных. Позже вы можете решить, что список вообще не нужен. Обновленная структура позволит легко изменить код. Например, если изменить реализацию, чтобы в ней использовались связанные кортежи, пользователь класса `Stack` этого даже не заметит:

```
class Stack:  
    def __init__(self):  
        self._items = None  
        self._size = 0  
  
    def push(self, item):  
        self._items = (item, self._items)  
        self._size += 1  
  
    def pop(self):  
        (item, self._items) = self._items  
        self._size -= 1  
        return item  
  
    def __len__(self):  
        return self._size
```

Чтобы решить, применять наследование или нет, остановитесь и спросите себя: действительно ли создаваемый вами объект — это специализированная версия родительского класса или вы просто используете его как компонент для построения чего-то другого? В последнем случае наследование лучше не применять.

7.9. ЗАМЕНА НАСЛЕДОВАНИЯ ФУНКЦИЯМИ

Иногда мы пишем классы с единственным методом, который должен адаптироваться для разных случаев. Например, вы пишете следующий класс разбора данных:

```
class DataParser:
    def parse(self, lines):
        records = []
        for line in lines:
            row = line.split(',')
            record = self.make_record(row)
            records.append(record)
        return records

    def make_record(self, row):
        raise NotImplementedError()

class PortfolioDataParser(DataParser):
    def make_record(self, row):
        return {
            'name': row[0],
            'shares': int(row[1]),
            'price': float(row[2])
        }

parser = PortfolioDataParser()
data = parser.parse(open('portfolio.csv'))
```

Здесь слишком много служебного кода. Если вы пишете много классов всего с одним методом, подумайте, нельзя ли заменить их функциями:

```
def parse_data(lines, make_record):
    records = []
    for line in lines:
        row = line.split(',')
        record = make_record(row)
        records.append(record)
    return records

def make_dict(row):
    return {
        'name': row[0],
        'shares': int(row[1]),
        'price': float(row[2])
    }

data = parse_data(open('portfolio.csv'), make_dict)
```

Этот код намного проще и не уступает прошлой версии в гибкости, а простые функции легче в тестировании. При необходимости его всегда можно будет расширить в классы. Преждевременная абстракция обычно нежелательна.

7.10. ДИНАМИЧЕСКАЯ И УТИНАЯ ТИПИЗАЦИИ

Динамическая типизация (или динамическое связывание) — механизм времени выполнения в Python для поиска атрибутов объектов. Именно она позволяет Python работать с экземплярами без ограничений по типу. В Python с именами переменных не связывается тип. Поэтому процесс связывания атрибутов не зависит от разновидности объекта, представленного переменной `obj`. Если вы обращаетесь к атрибуту (например, `obj.name`), он будет работать с любым объектом `obj`, содержащим `name`. Такое поведение иногда называется утиной типизацией. Здесь имеется в виду поговорка «Если что-то выглядит, крикает и ходит как утка, то это, вероятно, и есть утка».

В Python часто пишут программы, зависящие от этого поведения. Например, для создания улучшенной версии существующего объекта вы можете или наследовать его, или создать совершенно новый, который выглядит и ведет себя так же, но в остальном не связан с ним. Последний подход часто используется для обеспечения слабого связывания компонентов программы. Например, код может быть написан для работы с любой разновидностью объекта, если он определяет некоторый набор методов. Один из самых распространенных примеров такого кода встречается с разными итерируемыми объектами, определенными в стандартной библиотеке. Есть много объектов, работающих в циклах `for` для производства значений: списки, файлы, генераторы, строки и т. д. Но ни один из них не унаследован от какого-нибудь специального базового класса `Iterable`. Они лишь реализуют методы для выполнения перебора — и все работает.

7.11. ОПАСНОСТЬ НАСЛЕДОВАНИЯ ОТ ВСТРОЕННЫХ ТИПОВ

Python допускает наследование от встроенных типов. Но оно сопряжено с определенным риском. Например, если вы решили субклассировать `dict`, чтобы принудительно использовать ключи в верхнем регистре, можно переопределить метод `__setitem__()` так:

```
class udict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key.upper(), value)
```

И на первый взгляд такое решение работает:

```
>>> u = udict()
>>> u['name'] = 'Guido'
>>> u['number'] = 37
>>> u
{ 'NAME': 'Guido', 'NUMBER': 37 }
```

Но потом выясняется, что это не так — вам только казалось, что класс работает. И теперь начинает казаться, что он вообще не работает:

```
>>> u = udict(name='Guido', number=37)
>>> u
{ 'name': 'Guido', 'number': 37 }
>>> u.update(color='blue')
>>> u
{ 'name': 'Guido', 'number': 37, 'color': 'blue' }
```

Проблема в том, что встроенные типы Python не реализуются как нормальные классы Python — они написаны на C. Большинство методов работают в мире C. Например, `dict.update()` напрямую манипулирует данными словаря в обход переопределенного метода `__setitem__()` из вашего класса `udict`.

В модуле `collections` есть специальные классы `UserDict`, `UserList` и `UserString`. Они могут использоваться для создания безопасных субклассов `dict`, `list` и `str`. Следующее решение работает гораздо лучше:

```
from collections import UserDict

class udict(UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key.upper(), value)
```

Пример использования новой версии:

```
>>> u = udict(name='Guido', num=37)
>>> u.update(color='Blue')
>>> u
{ 'NAME': 'Guido', 'NUM': 37, 'COLOR': 'Blue' }
>>> v = udict(u)
>>> v['title'] = 'BDFL'
>>> v
{ 'NAME': 'Guido', 'NUM': 37, 'COLOR': 'Blue', 'TITLE': 'BDFL' }
```

Обычно субклассирования встроенных типов лучше избегать. Например, при построении новых контейнеров лучше создать новый класс, как для класса `Stack` в разделе 7.8. Если вам действительно нужно субклассировать встроенный тип, то приготовьтесь, что работы будет больше, чем вам казалось изначально.

7.12. ПЕРЕМЕННЫЕ И МЕТОДЫ КЛАССА

В определении класса предполагается, что все функции работают с экземпляром, который всегда передается в первом параметре `self`. Но сам класс тоже представлен объектом, который может нести состояние и оперировать им. Например, можно подсчитывать число созданных экземпляров класса в переменной `num_accounts`:

```
class Account:
    num_accounts = 0

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance
        Account.num_accounts += 1

    def __repr__(self):
        return f'{type(self).__name__}({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.deposit(-amount) # Must use self.deposit()

    def inquiry(self):
        return self.balance
```

Переменные класса определяются за пределами обычного метода `__init__()`. Для их изменения нужно использовать класс, а не `self`:

```
>>> a = Account('Guido', 1000.0)
>>> b = Account('Eva', 10.0)
>>> Account.num_accounts
2
>>>
```

К переменным класса можно обращаться и через экземпляры, хотя это и не-много необычно:

```
>>> a.num_accounts
2
```

```
>>> c = Account('Ben', 50.0)
>>> Account.num_accounts
3
>>> a.num_accounts
3
>>>
```

Это работает, потому что поиск атрибутов в экземплярах проверяет связанный класс, если в самом экземпляре нет соответствующего атрибута. Этот же механизм используется Python при поиске методов.

Также возможно определить и *методы класса*. Они применяются к самому классу, а не к экземплярам. Обычно методы класса используются для определения альтернативных конструкторов экземпляров. Представьте, что в требованиях была оговорена возможность создания экземпляров `Account` по унаследованному формату данных корпоративного уровня:

```
data = '''
<account>
  <owner>Guido</owner>
  <amount>1000.0</amount>
</account>
'''
```

Для этого можно написать метод класса:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    @classmethod
    def from_xml(cls, data):
        from xml.etree.ElementTree import XML
        doc = XML(data)
        return cls(doc.findtext('owner'), float(doc.findtext('amount')))

# Пример использования
data = '''
<account>
  <owner>Guido</owner>
  <amount>1000.0</amount>
</account>
'''

a = Account.from_xml(data)
```

В первом аргументе метода класса всегда передается сам класс. Этот аргумент часто называется `cls`. Здесь `cls` присваивается `Account`. Если цель метода

класса — создание нового экземпляра, для этого должны быть предприняты явные шаги. В последней строке примера вызов `cls(..., ...)` аналогичен вызову `Account(..., ...)` с двумя аргументами.

Переданный в аргументе класс решает важную проблему, связанную с наследованием. Допустим, вы определили производный от `Account` класс и теперь хотите создать экземпляр этого класса. Оказывается, и этот способ работает:

```
class EvilAccount(Account):
    pass

e = EvilAccount.from_xml(data) # Создает 'EvilAccount'
```

Почему код работает? Дело в том, что `EvilAccount` теперь передается как `cls`. Поэтому последняя команда метода класса `from_xml()` теперь создает экземпляр `EvilAccount`.

Переменные и методы класса иногда используются совместно для настройки и управления поведением экземпляров. Рассмотрим другой пример — класс `Date`:

```
import time

class Date:
    datefmt = '{year}-{month:02d}-{day:02d}'
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __str__(self):
        return self.datefmt.format(year=self.year,
                                    month=self.month,
                                    day=self.day)

    @classmethod
    def from_timestamp(cls, ts):
        tm = time.localtime(ts)
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)

    @classmethod
    def today(cls):
        return cls.from_timestamp(time.time())
```

Здесь есть переменная класса `datefmt`, настраивающая вывод метода `__str__()`. Для модификации вывода можно использовать наследование:

```

class MDYDate(Date):
    datefmt = '{month}/{day}/{year}'

class DMYDate(Date):
    datefmt = '{day}/{month}/{year}'

# Пример
a = Date(1967, 4, 9)
print(a) # 1967-04-09

b = MDYDate(1967, 4, 9)
print(b) # 4/9/1967

c = DMYDate(1967, 4, 9)
print(c) # 9/4/1967

```

Такая настройка конфигурации с использованием переменных класса и наследования — популярный механизм для настройки поведения экземпляров. Использование методов класса имеет решающее значение для его работы, ведь они гарантируют создание объекта правильного типа:

```

a = MDYDate.today()
b = DMYDate.today()
print(a)      # 2/13/2019
print(b)      # 13/2/2019

```

Альтернативное конструирование экземпляров — самое частое применение методов классов. В популярной схеме выбора имен таких методов используется префикс `from_`, например `from_timestamp()`. Эта схема встречается в методах класса в стандартной библиотеке и в сторонних пакетах. Например, у словарей есть метод класса для создания заранее инициализированного словаря по множеству ключей:

```

>>> dict.from_keys(['a','b','c'], 0)
{'a': 0, 'b': 0, 'c': 0}
>>>

```

Предупреждение: Python не управляет методами класса в пространстве имен, отдельном от методов экземпляра. Поэтому они все еще могут вызываться для экземпляров:

```

d = Date(1967,4,9)
b = d.today()      # Вызывает Date.now(Date)

```

Это может привести к путанице, потому что вызов `d.today()` не имеет никакого отношения к экземпляру `d`. Но вы можете обнаружить `today()` среди методов экземпляров `Date` в IDE и в документации.

7.13. СТАТИЧЕСКИЕ МЕТОДЫ

Иногда класс просто используется как пространство имен для функций, объявленных как статические методы с использованием `@staticmethod`. В отличие от обычного метода или метода класса, статический не получает дополнительный аргумент `self` или `cls`. Это обычная функция, которая определяется внутри класса:

```
class Ops:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def sub(x, y):
        return x - y
```

Экземпляры таких классов обычно не создаются. Вместо этого используется прямой вызов функций через класс:

```
a = Ops.add(2, 3) # a = 5
b = Ops.sub(4, 5) # a = -1
```

Иногда другие классы будут использовать набор статических методов вроде этого для реализации заменяемого/настраиваемого поведения или как что-то, слабо имитирующее поведение модуля импорта. Рассмотрим применение наследования в прошлом примере `Account`:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'{type(self).__name__}({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def inquiry(self):
        return self.balance

# Специальная разновидность Account
class EvilAccount(Account):
```

```

def deposit(self, amount):
    self.balance += 0.95 * amount

def inquiry(self):
    if random.randint(0,4) == 1:
        return 1.10 * self.balance
    else:
        return self.balance

```

Применение наследования здесь выглядит немного странно. Оно вводит две разновидности объектов — `Account` и `EvilAccount`. Нет очевидного способа изменить `Account` на `EvilAccount` или обратно, ведь это связано с изменением типа экземпляра. Возможно, лучше воплотить специфику `EvilAccount` как политику `Account`. Вот альтернативная формулировка `Account`, решающая эту проблему с помощью статических методов:

```

class StandardPolicy:
    @staticmethod
    def deposit(account, amount):
        account.balance += amount

    @staticmethod
    def withdraw(account, amount):
        account.balance -= amount

    @staticmethod
    def inquiry(account):
        return account.balance

class EvilPolicy(StandardPolicy):
    @staticmethod
    def deposit(account, amount):
        account.balance += 0.95*amount

    @staticmethod
    def inquiry(account):
        if random.randint(0,4) == 1:
            return 1.10 * account.balance
        else:
            return account.balance

class Account:
    def __init__(self, owner, balance, *, policy=StandardPolicy):
        self.owner = owner
        self.balance = balance
        self.policy = policy

    def __repr__(self):

```

```

        return f'Account({self.policy}, {self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.policy.deposit(self, amount)

    def withdraw(self, amount):
        self.policy.withdraw(self, amount)

    def inquiry(self):
        return self.policy.inquiry(self)

```

Здесь создается только один тип экземпляра — `Account`. Но он содержит специальный атрибут `policy`, обеспечивающий реализацию разных методов. При необходимости политику для `Account` можно изменить динамически:

```

>>> a = Account('Guido', 1000.0)
>>> a.policy
<class 'StandardPolicy'>
>>> a.deposit(500)
>>> a.inquiry()
1500.0
>>> a.policy = EvilPolicy
>>> a.deposit(500)
>>> a.inquiry()      # Может применяться случайный множитель 1.10x
1975.0
>>>

```

Одна из причин, по которым применение `@staticmethod` имеет смысл, в том, что можно не создавать экземпляры `StandardPolicy` или `EvilPolicy`. Главное предназначение этих классов — организация набора методов, а не хранение дополнительных данных экземпляра, связанных с `Account`. Но специфическая природа Python со слабым связыванием позволяет обновить политику, так чтобы она содержала собственные данные. Замените статические обычными методами экземпляра:

```

class EvilPolicy(StandardPolicy):
    def __init__(self, deposit_factor, inquiry_factor):
        self.deposit_factor = deposit_factor
        self.inquiry_factor = inquiry_factor

    def deposit(self, account, amount):
        account.balance += self.deposit_factor * amount

    def inquiry(self, account):
        if random.randint(0,4) == 1:
            return self.inquiry_factor * account.balance
        else:

```

```
return account.balance
```

```
# Пример использования
```

```
a = Account('Guido', 1000.0, policy=EvilPolicy(0.95, 1.10))
```

Такой подход — стандартная стратегия реализации конечных автоматов и похожих объектов. Каждое рабочее состояние внедряется в отдельный класс с методами (часто статическими). Изменяемая переменная экземпляра (атрибут `policy` в этом примере) может использоваться для хранения подробностей реализации, относящихся к текущему рабочему состоянию.

7.14. О ПАТТЕРНАХ ПРОЕКТИРОВАНИЯ

При написании объектно-ориентированных программ многие стремятся реализовать известные паттерны проектирования: «стратегия», «приспособленец», «одиночка» и т. д. Большинство из них происходят из знаменитой книги «Паттерны проектирования» Эрика Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса¹. Если вам знакомы эти паттерны, общие принципы проектирования в других языках могут быть применены и в Python. Но многие документированные шаблоны предназначены для решения конкретных проблем, возникающих из-за строгой статической системы типов C++ или Java. Динамическая природа Python делает многие из этих шаблонов устаревшими, излишними или просто ненужными.

Есть ряд универсальных принципов написания хорошего кода. Например, стремление писать код простой в отладке, тестировании и расширении. Такие базовые стратегии, как написание классов с полезными методами `__repr__()`, предпочтение композиции перед наследованием и разрешение внедрения зависимостей, очень важны для достижения этих целей.

Программисты Python предпочитают работать с кодом, который называют питоническим. Это значит, что объекты соблюдают разные встроенные протоколы (перебор, контейнеры или управление контекстом). Например, вместо того чтобы пытаться реализовать какой-то экзотический шаблон обхода данных из книги по программированию на Java, программист Python реализует его с помощью функции-генератора или просто заменит весь шаблон несколькими поисками по словарю.

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны проектирования. — СПб.: Питер, 2021.

7.15. ИНКАПСУЛЯЦИЯ ДАННЫХ И ПРИВАТНЫЕ АТТРИБУТЫ

В языке Python все атрибуты и методы классов *открыты* — доступны без ограничений. Такой подход часто нежелателен в объектно-ориентированных приложениях, где есть причины для сокрытия или инкапсуляции внутренних подробностей реализации.

Для решения этой проблемы Python использует соглашения об именах как средство обозначения предполагаемого использования. Одно из них гласит, что имена, начинающиеся с одного символа подчеркивания (`_`), относятся к внутренней реализации. Например, в следующей версии класса `Account` атрибут `balance` преобразован в приватный:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self._balance!r})'

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def inquiry(self):
        return self._balance
```

Здесь атрибут `_balance` рассматривается как часть внутренней реализации. Ничто не мешает пользователю обратиться к нему напрямую. Но начальный символ подчеркивания настойчиво указывает на то, что пользователю следует поискать интерфейс для внешнего использования, например метод `Account.inquiry()`.

Остается разобраться, доступны ли внутренние атрибуты для использования в subclasses. Сможет ли прошлый пример наследования напрямую обратиться к атрибуту `_balance` своего родителя?

```
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return 1.10 * self._balance
        else:
            return self._balance
```

Это считается приемлемым в Python. Скорее всего, IDE и другие инструменты будут предоставлять доступ к таким атрибутам. Если у вас есть опыт программирования на C++, Java или других похожих объектно-ориентированных языках, `_balance` можно рассматривать как аналог защищенного (`protected`) атрибута.

Чтобы усилить приватность атрибута, снабдите имя префиксом из двух начальных символов подчеркивания (`__`). Все имена вида `__имя` автоматически преобразуются в новое имя по схеме `__Класс__имя`. Так гарантируется, что приватные имена, используемые в суперклассе, не будут замещены такими же в производном классе:

```
class A:
    def __init__(self):
        self.__x = 3 # Преобразуется в self._A__x

    def __spam(self): # Преобразуется в _A__spam()
        print('A.__spam', self.__x)

    def bar(self):
        self.__spam() # Вызывается только A.__spam()

class B(A):
    def __init__(self):
        A.__init__(self)
        self.__x = 37 # Преобразуется в self._B__x

    def __spam(self): # Преобразуется в _B__spam()
        print('B.__spam', self.__x)

    def grok(self):
        self.__spam() # Вызывает B.__spam()
```

В этом примере используются два разных присваивания атрибуту `__x`. Может показаться, что класс `B` пытается переопределить метод `__spam()` наследованием. Но это не так. Преобразование дает уникальные имена, которые используются в каждом определении. Попробуйте выполнить следующий пример:

```
>>> b = B()
>>> b.bar()
A.__spam 3
>>> b.grok()
B.__spam 37
>>>
```

Чтобы увидеть преобразованные имена, просмотрите список переменных экземпляра:


```
>>> vars(b)
{ '_A__x': 3, '_B__x': 37 }
>>> b._A__spam()
A.__spam 3
>>> b._B__spam
B.__spam 37
>>>
```

Эта схема создает иллюзию сокрытия данных, но нет реального механизма, который бы блокировал доступ к приватным атрибутам класса. Если имена класса и соответствующего приватного атрибута известны, к ним можно обратиться по преобразованному имени. Если доступ к приватным атрибутам создает проблемы, попробуйте применить более жесткий процесс рецензирования кода.

На первый взгляд преобразование имен кажется лишним шагом при обработке кода. Но оно выполняется всего один раз при определении класса и не требует лишних ресурсов при работе программы.

Учтите: преобразование имен не выполняется в таких функциях, как `getattr()`, `hasattr()`, `setattr()` или `delattr()`, где имя атрибута задается в виде строки. В таких функциях для обращения к атрибуту придется использовать преобразованное имя вида `'_Класс__имя'`.

На практике лучше не задумываться о конфиденциальности имен. Имена с одиночным символом подчеркивания встречаются часто, с двойным — реже. Вы можете предпринять дополнительные меры для сокрытия атрибутов. Но результат вряд ли окупит лишние усилия и возрастание сложности. Самое полезное, что нужно запомнить по этому поводу: если вы видите начальные подчеркивания в имени, то это почти наверняка какая-то подробность внутренней реализации, от которой лучше держаться подальше.

7.16. АННОТАЦИИ ТИПОВ

У атрибутов классов, определяемых пользователем, нет ограничений по типу или значению. Вы можете присвоить атрибуту любое значение:

```
>>> a = Account('Guido', 1000.0)
>>> a.owner
'Guido'
>>> a.owner = 37
>>> a.owner
37
>>> b = Account('Eva', 'a lot')
>>> b.deposit(' more')
```

```
>>> b.inquiry()
'a lot more'
>>>
```

Если для вас это проблема, есть несколько возможных решений. Самое простое — не делайте этого! Другое решение основано на внешних инструментах (статических анализаторах и средствах проверки типов). Для них классы поддерживают возможность задавать необязательные аннотации типов для некоторых атрибутов:

```
class Account:
    owner: str          # Аннотация типа
    _balance: float     # Аннотация типа
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance
    ...
```

Включение аннотаций типов ничего не меняет в фактическом поведении класса во время выполнения — никакие дополнительные проверки не выполняются, и ничто не мешает пользователю задать неверные значения в своем коде. Но аннотации могут дать пользователю более полезную информацию в редакторе, предотвращая возможные опечатки.

На практике с указанием точных аннотаций типов могут возникнуть сложности. Позволяет ли класс `Account` использовать `int` вместо `float`? А как насчет `Decimal`? Оказывается, все эти типы работают, хотя аннотация свидетельствует об обратном.

```
from decimal import Decimal

a = Account('Guido', Decimal('1000.0'))
a.withdraw(Decimal('50.0'))
print(a.inquiry())           # -> 950.0
```

Знание того, как правильно организовать типы в таких ситуациях, выходит за рамки этой книги. Если у вас возникнут сомнения, лучше не продолжать (разве что вы активно пользуетесь средствами проверки типов в вашем коде).

7.17. СВОЙСТВА

Как упоминалось в прошлом разделе, Python не устанавливает никаких ограничений времени выполнения для значений или типов атрибутов. Но такие ограничения возможны. Для этого нужно поместить атрибут под управление

свойства. Это разновидность атрибута, которая перехватывает обращения к нему и обрабатывает их методами, определенными пользователем. Такие методы могут управлять атрибутом так, как считают нужным:

```
import string

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance

    @property
    def owner(self):
        return self._owner

    @owner.setter
    def owner(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected str')
        if not all(c in string.ascii_uppercase for c in value):
            raise ValueError('Must be uppercase ASCII')
        if len(value) > 10:
            raise ValueError('Must be 10 characters or less')
        self._owner = value
```

Здесь атрибут `owner` ограничивается строкой из 10 символов верхнего регистра в кодировке ASCII. Вот, как это работает при использовании класса:

```
>>> a = Account('GUIDO', 1000.0)
>>> a.owner = 'EVA'
>>> a.owner = 42
Traceback (most recent call last):
...
TypeError: Expected str
>>> a.owner = 'Carol'
Traceback (most recent call last):
...
ValueError: Must be uppercase ASCII
>>> a.owner = 'RENÉE'
Traceback (most recent call last):
...
ValueError: Must be uppercase ASCII
>>> a.owner = 'RAMAKRISHNAN'
Traceback (most recent call last):
...
ValueError: Must be 10 characters or less
>>>
```

`@property` помечает атрибут как свойство. Здесь он применяется к атрибуту `owner`. Этот декоратор всегда применяется к методу, получающему значение атрибута. В этом примере метод возвращает фактическое значение, которое сохраняется в приватном атрибуте `_owner`. Декоратор `@owner.setter` используется для необязательной реализации метода, присваивающего значение атрибута. Этот метод выполняет проверки типа и значения перед сохранением значения в `_owner`.

Важнейшая особенность свойств в том, что связанное с ними имя (как `owner` в этом примере) становится «волшебным»: любое использование этого атрибута автоматически направляется через реализованные вами методы чтения/записи. Вам не придется изменять код, чтобы эта схема заработала. Не нужно вносить изменения в метод `Account.__init__()`. Вас это может удивить, ведь `__init__()` выполняет присваивание `self.owner = owner` вместо использования приватного атрибута `self._owner`. Это сделано специально: свойство `owner` вводилось именно для проверки значений атрибута, что определенно нужно делать при создании экземпляров. Вы увидите, что все работает как предполагалось:

```
>>> a = Account('Guido', 1000.0)
Traceback (most recent call last):
  File "account.py", line 5, in __init__
    self.owner = owner
  File "account.py", line 15, in owner
    raise ValueError('Must be uppercase ASCII')
ValueError: Must be uppercase ASCII
>>>
```

При каждом обращении к атрибуту свойства автоматически вызывается метод, поэтому реальное значение должно храниться под другим именем. Вот почему внутри методов чтения и записи используется имя `_owner`. `owner` не может использоваться для хранения — это приведет к бесконечной рекурсии.

Как правило, свойства позволяют перехватывать любое конкретное имя атрибута. Вы можете реализовать методы для чтения, записи или удаления значения атрибута:

```
class SomeClass:
    @property
    def attr(self):
        print('Getting')

    @attr.setter
    def attr(self, value):
```

```

        print('Setting', value)

    @attr.deleter
    def attr(self):
        print('Deleting')

# Пример
s = SomeClass()
s.attr      # Чтение
s.attr = 13  # Запись
del s.attr   # Удаление

```

Реализовать все части свойства необязательно. На самом деле свойства часто используются для реализации атрибутов вычисляемых данных, доступных только для чтения:

```

class Box(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

    @property
    def perimeter(self):
        return 2*self.width + 2*self.height

# Пример использования
b = Box(4, 5)
print(b.area)      # -> 20
print(b.perimeter) # -> 18
b.area = 5         # Ошибка: невозможно задать атрибут

```

При проектировании класса нужно стремиться к созданию как можно более унифицированного интерфейса. Без свойств к некоторым значениям можно будет обращаться как к простым атрибутам (`b.width` или `b.height`), а к другим значениям вы будете обращаться как к методам (`b.area()` или `b.perimeter()`). Пользователю придется следить за добавлением круглых скобок `()`, что приводит к путанице. Свойства помогут справиться с этой проблемой.

Программисты Python часто не осознают, что сами методы неявно обрабатываются как свойства. Возьмем следующий класс:

```

class SomeClass:
    def yow(self):
        print('Yow!')

```

Когда пользователь создает экземпляр (например, `s = SomeClass()`) и обращается к `s.yow`, исходный объект функции `yow` не возвращается. Вместо этого вы получаете связанный метод:

```
>>> s = SomeClass()
>>> s.yow
<bound method SomeClass.yow of <__main__.SomeClass object at
0x10e2572b0>>
>>>
```

Как это произошло? У поведения функций, включенных в класс, много общего с поведением свойств. Функции перехватывают обращения к атрибутам и «за кулисами» создают связанные методы. Определяя статические методы и методы класса с помощью `@staticmethod` и `@classmethod`, вы изменяете этот процесс.

`@staticmethod` возвращает метод как есть, без дополнительной обработки. Подробнее об этом можно узнать в разделе 7.28.

7.18. ТИПЫ, ИНТЕРФЕЙСЫ И АБСТРАКТНЫЕ БАЗОВЫЕ КЛАССЫ

При создании экземпляра класса его тип — это сам класс. Для проверки принадлежности к классу используется встроенная функция `isinstance(obj, cls)`. Она возвращает `True`, если объект `obj` принадлежит `cls` или любому классу, производному от `cls`:

```
class A:
    pass

class B(A):
    pass

class C:
    pass

a = A()          # Экземпляр 'A'
b = B()          # Экземпляр 'B'
c = C()          # Экземпляр 'C'
type(a)          # Возвращает объект класса A
isinstance(a, A) # Возвращает True
isinstance(b, A) # Возвращает True, класс B является производным от A
isinstance(b, C) # Возвращает False, класс B не является производным от C
```

Точно так же встроенная функция `issubclass(A, B)` возвращает `True`, если класс `A` — субкласс класса `B`:

```
issubclass(B, A) # Возвращает True
issubclass(C, A) # Возвращает False
```

Отношения типов между классами часто используются как основа спецификаций программных интерфейсов. Например, базовый класс верхнего уровня может быть реализован для задания требований программного интерфейса. Затем он может использоваться для аннотаций типов или их защитной проверки вызовом `isinstance()`:

```
class Stream:
    def receive(self):
        raise NotImplementedError()

    def send(self, msg):
        raise NotImplementedError()

    def close(self):
        raise NotImplementedError()

# Пример
def send_request(stream, request):
    if not isinstance(stream, Stream):
        raise TypeError('Expected a Stream')
    stream.send(request)
    return stream.receive()
```

Такой код не предполагает прямое использование `Stream`. Вместо этого разные классы наследуют от `Stream` и реализуют нужную функциональность. Пользователь создает экземпляр одного из них:

```
class SocketStream(Stream):
    def receive(self):
        ...

    def send(self, msg):
        ...

    def close(self):
        ...

class PipeStream(Stream):
    def receive(self):
        ...
```

```

    def send(self, msg):
        ...

    def close(self):
        ...
# Пример
s = SocketStream()
send_request(s, request)

```

Обратите внимание на контроль соблюдения интерфейса на стадии выполнения в `send_request()`. Не использовать ли вместо этого аннотацию типа?

```

# Указание интерфейса как аннотации типа
def send_request(stream:Stream, request):
    stream.send(request)
    return stream.receive()

```

Учитывая, что соблюдение аннотаций типа не контролируется, решение о том, как проверить аргумент на соответствие интерфейсу, зависит от времени проведения проверки — при выполнении в качестве этапа проверки кода или вообще никогда.

Такое применение классов интерфейсов чаще встречается при организации больших фреймворков и приложений. Но здесь важно убедиться, что субклассы действительно реализуют необходимый интерфейс.

Например, если субкласс решил не реализовывать один из обязательных методов или содержит простую опечатку, поначалу последствия могут быть незамеченными, так как код работает в стандартном случае. Но позднее программа аварийно завершится при вызове нереализованного метода. И конечно, это происходит только в полчетвертого ночи в приложении, уже запущенном в эксплуатацию.

Чтобы этого избежать, интерфейсы часто определяются в виде *абстрактных базовых классов* с использованием модуля `abc`. Он определяет базовый класс (`ABC`) и декоратор (`@abstractmethod`), которые используются совместно для описания интерфейса:

```

from abc import ABC, abstractmethod

class Stream(ABC):
    @abstractmethod
    def receive(self):
        pass

    @abstractmethod

```



```
def send(self, msg):
    pass

@abstractmethod
def close(self):
    pass
```

Абстрактный класс не предназначен для прямого создания экземпляров. Более того, при попытке создать экземпляр `Stream` происходит ошибка:

```
>>> s = Stream()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Stream with abstract methods
close, receive, send
>>>
```

В сообщении об ошибке указано, какие методы должны быть реализованы `Stream`. Это поможет вам в написании subclasses. Представьте, что вы допустили ошибку в написанном вами subclasse:

```
class SocketStream(Stream):

    def read(self): # Неправильное имя
        ...

    def send(self, msg):
        ...

    def close(self):
        ...
```

Абстрактный базовый класс обнаружит ее при создании экземпляра. Перехват ошибок на ранней стадии может быть очень полезен.

```
>>> s = SocketStream()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class SocketStream with abstract
methods receive
>>>
```

Хотя абстрактный класс нельзя создать, он может определять методы и свойства для использования в subclasses. Более того, абстрактные методы в базовом классе могут вызываться из них. Например, вызов `super().receive()` из subclasses разрешен.

7.19. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ, ИНТЕРФЕЙСЫ И ПРИМЕСИ

Python поддерживает множественное наследование. Если производный класс имеет более одного родителя, он наследует всю их функциональность:

```
class Duck:
    def walk(self):
        print('Waddle')

class Trombonist:
    def noise(self):
        print('Blat!')

class DuckBonist(Duck, Trombonist):
    pass

d = DuckBonist()
d.walk() # -> Waddle
d.noise() # -> Blat!
```

Концептуально это отличная идея, но затем начинают проявляться практические реалии. Например, что произойдет, если каждый из классов `Duck` и `Trombonist` определит метод `__init__()`? Или если они оба определяют метод `noise()`? Становится ясно, что множественное наследование сопряжено с высоким риском.

Чтобы лучше понять фактическое использование множественного наследования, сделайте шаг назад и рассмотрите его как узкоспециализированный инструмент для организации и повторного использования кода, а не как метод программирования общего назначения. В частности, брать набор произвольных несвязанных классов и объединять их с множественным наследованием для создания странных объектов-мутантов — плохая идея. Никогда так не делайте.

Более распространенный случай множественного использования — организация отношений между типами и интерфейсами. В прошлом разделе была представлена концепция абстрактного базового класса. Он предназначен для определения программного интерфейса. У вас могут быть такие абстрактные классы:

```
from abc import ABC, abstractmethod

class Stream(ABC):
    @abstractmethod
```

```
def receive(self):
    pass

@abstractmethod
def send(self, msg):
    pass

@abstractmethod
def close(self):
    pass
class Iterable(ABC):
    @abstractmethod
    def __iter__(self):
        pass
```

С такими классами можно воспользоваться множественным наследованием для определения интерфейсов, реализованных производным классом:

```
class MessageStream(Stream, Iterable):
    def receive(self):
        ...
    def send(self):
        ...
    def close(self):
        ...
    def __iter__(self):
        ...
```

И снова множественное наследование ориентировано не на реализацию, а на отношения типов. В нашем случае никакие унаследованные методы ничего не делают. Повторно использовать код нельзя. Прежде всего отношения наследования позволяют выполнять такие проверки типов:

```
m = MessageStream()

isinstance(m, Stream)    # -> True
isinstance(m, Iterable) # -> True
```

Множественное наследование применяется и для определения *классов примесей* (mixins). Класс примеси изменяет или расширяет функциональность других классов. Возьмем следующие определения классов:

```
class Duck:
    def noise(self):
        return 'Quack'

    def waddle(self):
```

```

        return 'Waddle'

class Trombonist:
    def noise(self):
        return 'Blat!'

    def march(self):
        return 'Clomp'

class Cyclist:
    def noise(self):
        return 'On your left!'

    def pedal(self):
        return 'Pedaling'
```

Они никак не связаны друг с другом — между ними нет отношений наследования, они реализуют разные методы. Но есть и сходство: все они определяют метод `noise()`. Руководствуясь этим, можно определить следующие классы:

```

class LoudMixin:
    def noise(self):
        return super().noise().upper()

class AnnoyingMixin:
    def noise(self):
        return 3*super().noise()
```

Кажется, что с этими классами что-то не так. Они содержат всего один изолированный метод и используют `super()` для делегирования несуществующему родительскому классу. Эти классы даже не работают сами по себе:

```

>>> a = AnnoyingMixin()
>>> a.noise()
Traceback (most recent call last):
...
AttributeError: 'super' object has no attribute 'noise'
>>>
```

Дело в том, что перед вами классы примесей. Они могут работать только вместе с другими классами, реализующими недостающую функциональность:

```

class LoudDuck(LoudMixin, Duck):
    pass

class AnnoyingTrombonist(AnnoyingMixin, Trombonist):
    pass
```

```

class AnnoyingLoudCyclist(AnnoyingMixin, LoudMixin, Cyclist):
    pass

d = LoudDuck()
d.noise() # -> 'QUACK'

t = AnnoyingTrombonist()
t.noise() # -> 'Blat!Blat!Blat!'

c = AnnoyingLoudCyclist()
c.noise() # -> 'ON YOUR LEFT!ON YOUR LEFT!ON YOUR LEFT!'

```

Так как классы примесей определяются точно так же, как и обычные, желательно включать слово *Mixin* в их имена. Это соглашение об именах более четко выражает предназначение класса.

Чтобы понять принцип работы примесей, нужно чуть больше знать о том, как работают наследование и функция `super()`.

Во-первых, при использовании наследования Python строит линейную цепочку классов, называемую MRO (Method Resolution Order). Она доступна в атрибуте `__mro__` класса.

Несколько примеров одиночного наследования:

```

class Base:
    pass

class A(Base):
    pass

class B(A):
    pass

Base.__mro__ # -> (<class 'Base'>, <class 'object'>)
A.__mro__   # -> (<class 'A'>, <class 'Base'>, <class 'object'>)
B.__mro__   # -> (<class 'B'>, <class 'A'>, <class 'Base'>, <class
'object'>)

```

MRO определяет порядок поиска атрибутов. Когда вы ищете атрибут в экземпляре класса, все классы из MRO проверяются в указанном порядке. Поиск останавливается при обнаружении первого совпадения. Класс `object` есть в MRO, потому что все классы унаследованы от `object` независимо от того, указан ли он в качестве родителя.

Для поддержки множественного наследования Python реализует так называемое кооперативное множественное наследование. При нем все классы

включаются в MRO по двум основным правилам упорядочения: первое — производный класс всегда должен проверяться до любых его родителей, второе — если у класса несколько родителей, они должны проверяться в порядке, в котором они перечисляются в списке наследования потомка. В основном эти правила создают полезный MRO. Но точный алгоритм упорядочения классов сложен, и не основан на простых решениях типа обхода в глубину или в ширину. Вместо этого порядок определяется алгоритмом C3-линеаризации, описанным в статье *A Monotonic Superclass Linearization for Dylan* (К. Barrett и др., представлена на конференции OOPSLA'96). Неочевидный аспект этого алгоритма — некоторые иерархии классов отклоняются Python с ошибкой `TypeError`:

```
class X: pass
class Y(X): pass
class Z(X,Y): pass # TypeError.
                  # Не удастся создать целостный список MRO
```

В этом случае алгоритм разрешения методов отвергает Z, так как не может определить разумный порядок базовых классов. Здесь X в списке наследования предшествует классу Y, поэтому он должен быть проверен первым. Но Y унаследован от X. И проверка X первым нарушает правило о том, что сначала должны проверяться потомки. На практике такие проблемы встречаются редко — и даже если встречаются, обычно свидетельствуют о более сложных проблемах проектирования.

В качестве примера практического использования MRO ниже приведен список MRO для класса `AnnoyingLoudCyclist` из предыдущего примера:

```
class AnnoyingLoudCyclist(AnnoyingMixin, LoudMixin, Cyclist):
    pass

AnnoyingLoudCyclist.__mro__
# (<class 'AnnoyingLoudCyclist'>, <class 'AnnoyingMixin'>,
# <class 'LoudMixin'>, <class 'Cyclist'>, <class 'object'>)
```

Здесь выполняются оба правила — каждый производный класс всегда предшествует своим родителям. `object` указан на последнем месте, так как он — родитель всех остальных классов. Родители перечисляются в порядке их следования в коде.

Поведение функции `super()` связано с нижележащим списком MRO. Она должна делегировать атрибуты следующему классу в MRO. Выбор зависит от класса, где используется `super()`. Например, когда `AnnoyingMixin` использует `super()`, он обращается к списку MRO экземпляра для определения

своей позиции. От этой точки он делегирует обращение к атрибуту следующему классу. В нашем примере при использовании `super().noise()` в классе `AnnoyingMixin` вызывается `LoudMixin.noise()`. Это объясняется тем, что `LoudMixin` — следующий класс, перечисленный в MRO для `AnnoyingLoudCyclist`. Операция `super().noise()` в классе `LoudMixin` делегируется классу `Cyclist`.

При любом использовании `super()` выбор следующего класса изменяется в зависимости от типа экземпляра. Например, если создать экземпляр `AnnoyingTrombonist`, то вызов `super().noise()` приведет к вызову `Trombonist.noise()`.

Проектирование для совместного множественного наследования и примесей — непростая задача. Вот несколько важных советов. Во-первых, производные классы всегда проверяются раньше любого базового класса в MRO. У примесей обычно есть общий родитель, который должен предоставить пустую реализацию методов. Если несколько классов примесей используются одновременно, они выстраиваются друг за другом. Общий родитель стоит на последнем месте и предоставляет реализацию по умолчанию или проверку ошибок:

```
class NoiseMixin:
    def noise(self):
        raise NotImplementedError('noise() not implemented')

class LoudMixin(NoiseMixin):
    def noise(self):
        return super().noise().upper()

class AnnoyingMixin(NoiseMixin):
    def noise(self):
        return 3 * super().noise()
```

Во-вторых, все реализации метода примеси должны иметь одинаковую сигнатуру. Одна из проблем с примесями в том, что они необязательны и часто непредсказуемо сочетаются. Для работы этой схемы нужно гарантировать, что операции с `super()` будут завершаться успешно независимо от следующего класса. Для этого все методы в цепочке должны иметь совместимую сигнатуру вызова.

В-третьих, важно проследить, чтобы везде использовался вызов `super()`. Иногда встречаются классы, которые обращаются к родителю с прямым вызовом:

```
class Base:
    def yow(self):
        print('Base.yow')
```

```

class A(Base):
    def yow(self):
        print('A.yow')
        Base.yow(self) # Прямой вызов метода родителя

class B(Base):
    def yow(self):
        print('B.yow')
        super().yow(self)

class C(A, B):
    pass

c = C()
c.yow()
# Вывод:
#   A.yow
#   Base.yow

```

Такие классы небезопасно использовать с множественным наследованием. Это нарушает правильную цепочку вызовов методов и приводит к путанице. В частности в примере выше вывода от `B.yow()` нет, хотя класс — часть иерархии наследования. При выполнении операций с множественным наследованием используйте `super()` вместо прямых вызовов методов в суперклассах.

7.20. ДИСПЕТЧЕРИЗАЦИЯ ВЫЗОВОВ В ЗАВИСИМОСТИ ОТ ТИПА

Иногда нужно написать код, вызывающий разные методы в зависимости от конкретного типа:

```

if isinstance(obj, Duck):
    handle_duck(obj)
elif isinstance(obj, Trombonist):
    handle_trombonist(obj)
elif isinstance(obj, Cyclist):
    handle_cyclist(obj)
else:
    raise RuntimeError('Unknown object')

```

Такие большие блоки `if-elif-else` неэлегантны и ненадежны. Часто встречается решение, основанное на диспетчеризации по словарю:

```

handlers = {
    Duck: handle_duck,

```



```

    Trombonist: handle_trombonist,
    Cyclist: handle_cyclist
}

```

Диспетчеризация

```

def dispatch(obj):
    func = handlers.get(type(obj))
    if func:
        return func(obj)
    else:
        raise RuntimeError(f'No handler for {obj}')

```

Это решение предполагает точное совпадение типа. Если в такой схеме диспетчеризации должно поддерживаться и наследование, потребуется проход по списку MRO:

```

def dispatch(obj):
    for ty in type(obj).__mro__:
        func = handlers.get(ty)
        if func:
            return func(obj)
    raise RuntimeError(f'No handler for {obj}')

```

Иногда диспетчеризация реализуется через интерфейс на базе классов с использованием `getattr()`:

```

class Dispatcher:
    def handle(self, obj):
        for ty in type(obj).__mro__:
            meth = getattr(self, f'handle_{ty.__name__}', None)
            if meth:
                return meth(obj)
        raise RuntimeError(f'No handler for {obj}')

    def handle_Duck(self, obj):
        ...

    def handle_Trombonist(self, obj):
        ...

    def handle_Cyclist(self, obj):
        ...

# Пример
dispatcher = Dispatcher()
dispatcher.handle(Duck())    # -> handle_Duck()
dispatcher.handle(Cyclist()) # -> handle_Cyclist()

```

Последний пример с использованием `getattr()` — довольно распространенный паттерн программирования.

7.21. ДЕКОРАТОРЫ КЛАССОВ

Иногда нужна дополнительная обработка после определения класса. Например, чтобы добавить класс в реестр или сгенерировать дополнительный вспомогательный код. Один из подходов основан на использовании декоратора класса. Это функция, которая получает класс на входе и возвращает на выходе. Ведение реестра может выглядеть так:

```
_registry = { }
def register_decoder(cls):
    for mt in cls.mimetypes:
        _registry[mt.mimetype] = cls
    return cls

# Фабричная функция, использующая реестр
def create_decoder(mimetype):
    return _registry[mimetype]()
```

Здесь функция `register_decoder()` ищет в классе атрибут `mimetypes`. Если он будет найден, то используется для включения класса в словарь, отображающий типы MIME с объектами классов. Для использования этой функции примените ее как декоратор перед определением класса:

```
@register_decoder
class TextDecoder:
    mimetypes = [ 'text/plain' ]

    def decode(self, data):
        ...

@register_decoder
class HTMLDecoder:
    mimetypes = [ 'text/html' ]
    def decode(self, data):
        ...

@register_decoder
class ImageDecoder:
    mimetypes = [ 'image/png', 'image/jpg', 'image/gif' ]
    def decode(self, data):
        ...

# Пример использования
decoder = create_decoder('image/jpg')
```

Декоратор может изменять содержимое полученного класса. Он может даже переписывать существующие методы. Это распространенная альтернатива классам примесей и множественному наследованию.

Рассмотрим следующие декораторы:

```
def loud(cls):
    orig_noise = cls.noise
    def noise(self):
        return orig_noise(self).upper()
    cls.noise = noise
    return cls

def annoying(cls):
    orig_noise = cls.noise
    def noise(self):
        return 3 * orig_noise(self)
    cls.noise = noise
    return cls

@annoying
@loud
class Cyclist(object):
    def noise(self):
        return 'On your left!'

    def pedal(self):
        return 'Pedaling'
```

Результат этого примера такой же, как и в случае с примесью из предыдущего раздела. Но в нем не используется ни множественное наследование, ни `super()`. Внутри каждого декоратора обращение к `cls.noise` выполняет ту же операцию, что и `super()`. Но это происходит только при применении декоратора (во время определения), поэтому итоговые вызовы `noise()` будут выполняться немного быстрее.

Декораторы классов могут использоваться и для создания совершенно нового кода. Например, при написании классов принято вводить содержательный метод `__repr__()` для улучшения отладки:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'{type(self).__name__}({self.x!r}, {self.y!r})'
```

Писать эти методы утомительно. Не сможет ли декоратор класса создать метод за вас?

```
import inspect
def with_repr(cls):
    args = list(inspect.signature(cls).parameters)
    argvals = ', '.join('{self.%s!r}' % arg for arg in args)
    code = 'def __repr__(self):\n'
    code += f' return f"{cls.__name__}({argvals})"\n'
    locs = { }
    exec(code, locs)
    cls.__repr__ = locs['__repr__']
    return cls

# Пример
@with_repr
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

В этом примере `__repr__()` генерируется по сигнатуре вызова метода `__init__()`. Метод создается в виде текстовой строки и передается `exec()` для создания функции. Эта функция присоединяется к классу.

Похожие приемы генерации кода используются во многих частях стандартной библиотеки. Например, есть удобный способ определения структур данных с использованием `@dataclass`:

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int
```

`@dataclass` автоматически создает такие методы, как `__init__()` и `__repr__()`, по аннотациям типов класса. Они создаются вызовом `exec()`, как в прошлом примере.

А вот так работает полученный класс `Point`:

```
>>> p = Point(2, 3)
>>> p
Point(x=2, y=3)
>>>
```

Один из недостатков такого подхода — низкая производительность при запуске. Динамическое создание кода вызовом `exec()` обходит оптимизации компилятора, которые Python обычно применяет к модулям. Определение большого количества классов таким способом может сильно замедлить импорт вашего кода.

Примеры из этого раздела иллюстрируют стандартные применения декораторов классов: регистрацию, переписывание и генерацию кода, проверку данных и т. д. Одна из проблем с декораторами классов в том, что они должны явно применяться к каждому классу, где они используются. Это не всегда хорошо. В следующем разделе описывается функция, позволяющая неявно манипулировать классами.

7.22. КОНТРОЛИРУЕМОЕ НАСЛЕДОВАНИЕ

Как вы поняли из прошлого раздела, иногда нужно определить класс и выполнить дополнительные действия. Декоратор класса — один из механизмов для решения этой задачи. Но родительский класс тоже может выполнять дополнительные действия от имени своих подклассов. Для этого нужно реализовать метод класса `__init_subclass__(cls)`:

```
class Base:
    @classmethod
    def __init_subclass__(cls):
        print('Initializing', cls)

# Пример (должно выводиться сообщение 'Initializing' для каждого класса)
class A(Base):
    pass

class B(A):
    pass
```

Если в классе есть метод `__init_subclass__()`, он выполняется автоматически при определении любого производного класса. Это происходит, даже если потомок скрыт где-то глубоко в иерархии наследования.

Методом `__init_subclass__()` могут выполняться многие операции, часто выполняемые с декораторами классов, например регистрация класса:

```
class DecoderBase:
    _registry = { }
    @classmethod
```

```

def __init_subclass__(cls):
    for mt in cls.mimetypes:
        DecoderBase._registry[mt.mimetype] = cls

# Фабричная функция, использующая реестр
def create_decoder(mimetype):
    return DecoderBase._registry[mimetype]()

class TextDecoder(DecoderBase):
    mimetypes = [ 'text/plain' ]
    def decode(self, data):
        ...

class HTMLDecoder(DecoderBase):
    mimetypes = [ 'text/html' ]
    def decode(self, data):
        ...

class ImageDecoder(DecoderBase):
    mimetypes = [ 'image/png', 'image/jpg', 'image/gif' ]
    def decode(self, data):
        ...

# Пример использования
decoder = create_decoder('image/jpg')
```

Пример класса, автоматически создающего `__repr__()` по сигнатуре `__init__()`:

```

import inspect

class Base:
    @classmethod
    def __init_subclass__(cls):
        # Создать метод __repr__
        args = list(inspect.signature(cls).parameters)
        argvals = ', '.join('{self.%s!r}' % arg for arg in args)
        code = 'def __repr__(self):\n'
        code += f'    return f"{cls.__name__}({argvals})"\n'
        locs = { }
        exec(code, locs)
        cls.__repr__ = locs['__repr__']

class Point(Base):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

При использовании множественного наследования вызывайте `super()` для гарантии того, что будут вызваны все классы, реализующие `__init_subclass__()`. Пример:

```
class A:
    @classmethod
    def __init_subclass__(cls):
        print('A.init_subclass')
        super().__init_subclass__()

class B:
    @classmethod
    def __init_subclass__(cls):
        print('B.init_subclass')
        super().__init_subclass__()

# Здесь должен появиться вывод из обоих классов
class C(A, B):
    pass
```

Контроль за наследованием через `__init_subclass__()` — одно из самых мощных средств настройки Python. Большая часть этой мощи связана с ее неявной природой. Базовый класс верхнего уровня может использовать это для незаметного наблюдения за всей иерархией дочерних классов. Такой контроль может регистрировать классы, переписывать методы, выполнять проверку данных и многое другое.

7.23. ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТОВ И УПРАВЛЕНИЕ ПАМЯТЬЮ

При определении класса полученный класс станет фабрикой для создания новых экземпляров:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

# Создание нескольких экземпляров Account
a = Account('Guido', 1000.0)
b = Account('Eva', 25.0)
```

Создание экземпляра выполняется за два шага с использованием специального метода `__new__()`, который создает новый экземпляр, и `__init__()`,

который его инициализирует. Операция `a = Account('Guido', 1000.0)` действует по следующей схеме:

```
a = Account.__new__(Account, 'Guido', 1000.0)
if isinstance(a, Account):
    Account.__init__('Guido', 1000.0)
```

Кроме первого аргумента, где передается класс вместо экземпляра, `__new__()` обычно получает те же аргументы, что и `__init__()`. Но реализация `__new__()`, используемая по умолчанию, просто игнорирует их. Иногда вы видите, как `__new__()` вызывается только с одним аргументом. Например, следующий код тоже работает:

```
a = Account.__new__(Account)
Account.__init__('Guido', 1000.0)
```

Прямой вызов метода `__new__()` нетипичен, но иногда используется для создания экземпляров с обходом вызова `__init__()`. Один из таких вариантов применения — методы классов:

```
import time

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        t = time.localtime()
        self = cls.__new__(cls) # Создание экземпляра
        self.year = t.tm_year
        self.month = t.tm_month
        self.day = t.tm_day
        return self
```

Модули, выполняющие сериализацию объектов (`pickle`), используют `__new__()` и для воссоздания экземпляров при десериализации объектов. При этом `__init__()` не вызывается.

Иногда класс определяет `__new__()` для изменения некоторых аспектов создания экземпляра. Типичные способы применения — кеширование экземпляров, одиночные (синглетные) экземпляры и неизменяемость. Допустим, вы хотите, чтобы класс `Date` выполнял интернирование дат (кеширование

и повторное использование экземпляров `Date` с идентичным годом, месяцем и днем). Одна из возможных реализаций выглядит так:

```
class Date:
    _cache = { }

    @staticmethod
    def __new__(cls, year, month, day):
        self = Date._cache.get((year, month, day))
        if not self:
            self = super().__new__(cls)
            self.year = year
            self.month = month
            self.day = day
            Date._cache[(year, month, day)] = self
        return self

    def __init__(self, year, month, day):
        pass

# Пример
d = Date(2012, 12, 21)
e = Date(2012, 12, 21)
assert d is e          # Тот же объект
```

В этом примере класс поддерживает внутренний словарь ранее созданных экземпляров `Date`. Во время создания нового экземпляра `Date` сначала проверяется содержимое кеша. При обнаружении совпадения возвращается найденный экземпляр. В противном случае создается и инициализируется новый.

Неочевидная подробность этого решения — пустой метод `__init__()`. Несмотря на то что экземпляры кешируются, при каждом вызове `Date()` все равно вызывается `__init__()`. Во избежание дублирования работы этот метод просто ничего не делает. Экземпляр создается в `__new__()` при первом создании экземпляра.

Лишнего вызова `__init__()` можно избежать, но для этого нужно изощриться. В одном из способов `__new__()` возвращает совершенно новый экземпляр типа, например принадлежащий другому классу. Другое решение основано на использовании метаклассов.

Созданными экземплярами управляет механизм подсчета ссылок. По достижении счетчиком ссылок нуля экземпляр немедленно уничтожается. Когда экземпляр должен быть уничтожен, интерпретатор сначала ищет метод `__del__()`, связанный с объектом, и вызывает его:

```

class Account(object):
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __del__(self):
        print('Deleting Account')
>>> a = Account('Guido', 1000.0)
>>> del a
Deleting Account
>>>

```

В какой-то момент программа использует команду `del` для удаления ссылки на объект. Если счетчик ссылок объекта достигает нуля, вызывается `__del__()`. Но, как правило, оператор `del` не вызывает напрямую `__del__()`, потому что в другом месте могут быть ссылки на другие объекты. Объекты могут удаляться иначе. Например, повторным присваиванием переменной или ее выходом из области видимости функции.

```

>>> a = Account('Guido', 1000.0)
>>> a = 42
Deleting Account
>>> def func():
...     a = Account('Guido', 1000.0)
...
>>> func()
Deleting Account
>>>

```

На практике необходимость в определении `__del__()` возникает нечасто. Исключения составляют случаи, когда уничтожение объекта требует дополнительных завершающих действий, например закрытия файла или сетевого подключения, освобождения других системных ресурсов. Даже в таких случаях лучше не полагаться на `__del__()` для корректного завершения, ведь нет гарантий, что этот метод будет вызван именно тогда, когда вы ожидаете. Для правильного освобождения ресурсов определите в объекте явный метод `close()` и обеспечьте поддержку классом протокола менеджера контекста, чтобы он мог использоваться с командой `with`. В следующем примере показаны все варианты:

```

class SomeClass:
    def __init__(self):
        self.resource = open_resource()

    def __del__(self):
        self.close()

```

```
def close(self):
    self.resource.close()

def __enter__(self):
    return self

def __exit__(self, ty, val, tb):
    self.close()

# Закрытие через __del__()
s = SomeClass()
del s

# Явное закрытие
s = SomeClass()
s.close()

# Закрытие в конце контекстного блока
with SomeClass() as s:
    ...
```

Еще раз подчеркну, что писать метод `__del__()` для класса почти всегда необязательно. В Python уже реализована сборка мусора, и заниматься ей не нужно, если только нет какой-то дополнительной операции, которая должна выполняться при уничтожении объекта. Но даже в этом случае может оказаться, что вызов `__del__()` необязателен, так как может оказаться, что объект уже запрограммирован на корректное завершение, даже в случае бездействия.

А если вам недостаточно проблем с подсчетом ссылок и уничтожением объекта, есть разные паттерны программирования (особенно те, где задействованы отношения «родитель — потомок», графы или кеширование), в которых объекты могут создавать *циклические ссылки*:

```
class SomeClass:
    def __del__(self):
        print('Deleting')

parent = SomeClass()
child = SomeClass()

# Создание циклических ссылок между родителем и потомком
parent.child = child
child.parent = parent

# Попытка удаления (вывод __del__ не появляется)
del parent
del child
```

В этом примере имена переменных уничтожаются, но никаких следов выполнения метода `__del__()` нет. В каждом из двух объектов есть внутренние ссылки на другой объект, поэтому счетчик ссылок не может уменьшиться до 0. Для таких ситуаций иногда должна выполняться специальная сборка мусора с обнаружением циклов. Со временем объекты будут уничтожены, но трудно предсказать, когда именно. Для форсированной сборки мусора вызовите метод `gc.collect()`. В модуле `gc` есть много других функций, связанных с циклическим сборщиком мусора и управлением памятью.

Из-за непредсказуемого хронометража сборки мусора для метода `__del__()` устанавливается ряд ограничений. Во-первых, любое исключение, распространяющееся из `__del__()`, выводится в `sys.stderr`, но в остальном игнорируется. Во-вторых, метод `__del__()` должен избегать таких операций, как захват блокировок и других ресурсов. Нарушение этого правила может привести к взаимной блокировке, если `__del__()` неожиданно сработает в середине выполнения другой, совершенно посторонней функции внутри седьмого внутреннего цикла обратных вызовов с обработкой сигналов и потоками. Чтобы определить метод `__del__()`, сделайте его простым.

7.24. СЛАБЫЕ ССЫЛКИ

Иногда объекты продолжают существовать, хотя вы бы хотели их уничтожить. В более раннем примере был показан класс `Date` с внутренним кешированием экземпляров. Один из недостатков такой реализации в том, что в ней не предусмотрена возможность удаления экземпляров из кеша. Поэтому он будет неограниченно разрастаться со временем.

Одно из возможных решений — создание слабой ссылки с использованием модуля `weakref`. Это позволит создать ссылку на объект без увеличения его счетчика. Для работы со слабой ссылкой нужно добавить лишний фрагмент кода, чтобы проверить наличие объекта, на который указывает ссылка.

Пример создания слабой ссылки:

```
>>> a = Account('Guido', 1000.0)
>>> import weakref
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x104617188; to 'Account' at 0x1046105c0>
>>>
```

В отличие от обычных ссылок, существование слабой допускает уничтожение исходного объекта:

```
>>> del a
>>> a_ref
<weakref at 0x104617188; dead>
>>>
```

Для получения объекта по слабой ссылке нужно вызвать ее как функцию без аргументов. Вы получите либо объект, на который она указывает, либо None. Пример:

```
acct = a_ref()
if acct is not None:
    acct.withdraw(10)

# Альтернатива
if acct := a_ref():
    acct.withdraw(10)
```

Слабые ссылки обычно используются вместе с кэшированием и другими простыми средствами управления памятью. Ниже приведена обновленная версия класса Date, которая автоматически удаляет объекты из кеша, если на него не осталось ни одной ссылки:

```
import weakref

class Date:
    _cache = { }

    @staticmethod
    def __new__(cls, year, month, day):
        selfref = Date._cache.get((year, month, day))
        if not selfref:
            self = super().__new__(cls)
            self.year = year
            self.month = month
            self.day = day
            Date._cache[(year, month, day)] = weakref.ref(self)
        else:
            self = selfref()
        return self

    def __init__(self, year, month, day):
        pass

    def __del__(self):
        del Date._cache[(self.year, self.month, self.day)]
```

Возможно, вы не сразу разберетесь в этом коде, но интерактивный сеанс поможет вам понять принцип его работы. Заметьте, что элемент, на который нет ни одной ссылки, удаляется из кеша:

```

>>> Date._cache
{}
>>> a = Date(2012, 12, 21)
>>> Date._cache
{(2012, 12, 21): <weakref at 0x10c7ee2c8; to 'Date' at 0x10c805518>}
>>> b = Date(2012, 12, 21)
>>> a is b
True
>>> del a
>>> Date._cache
{(2012, 12, 21): <weakref at 0x10c7ee2c8; to 'Date' at 0x10c805518>}
>>> del b
>>> Date._cache
{}
>>>

```

Как я уже говорил, метод `__del__()` класса вызывается, только когда счетчик ссылок объекта достигает нуля. Здесь первая команда `del a` уменьшает счетчик ссылок. Но так как на объект есть еще одна ссылка, он остается в кеше `Date._cache`. При удалении второго объекта вызывается `__del__()`, и кеш очищается.

Для поддержки слабых ссылок экземпляры должны иметь изменяемый атрибут `__weakref__`. Экземпляры определяемых пользователем классов обычно имеют такой атрибут по умолчанию. Но у встроенных типов и некоторых видов специальных структур данных — именованных кортежей, классов со `__slots__` — его нет. Чтобы создать слабые ссылки на эти типы, можно определить варианты с добавлением атрибута `__weakref__`:

```

class wdict(dict):
    __slots__ = ('__weakref__',)

w = wdict()
w_ref = weakref.ref(w)    # Теперь работает

```

Как вскоре будет показано, специальная переменная `__slots__` используется здесь для предотвращения ненужных затрат памяти.

7.25. ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ОБЪЕКТОВ И СВЯЗЫВАНИЕ АТРИБУТОВ

Состояние, связанное с экземпляром, хранится в словаре, доступном в атрибуте `__dict__` экземпляра. Этот словарь содержит данные, уникальные для каждого экземпляра:

```
>>> a = Account('Guido', 1100.0)
>>> a.__dict__
{'owner': 'Guido', 'balance': 1100.0}
```

Новые атрибуты могут добавляться в экземпляр в любой момент:

```
a.number = 123456 # Добавление атрибута 'number' в a.__dict__
a.__dict__['number'] = 654321
```

Изменения в экземпляре всегда отражаются в локальном атрибуте `__dict__`, если только он не находится под управлением свойства. И наоборот, при прямом внесении изменений в `__dict__` они будут отражаться в атрибутах.

Экземпляры связываются с классом через специальный атрибут `__class__`. Сам класс тоже служит тонкой прослойкой над словарем в `__dict__`. Именно в словаре класса находятся методы:

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
dict_keys(['__module__', '__init__', '__repr__', 'deposit', 'withdraw',
'inquiry', '__dict__', '__weakref__', '__doc__'])
>>> Account.__dict__['withdraw']
<function Account.withdraw at 0x108204158>
>>>
```

Классы связаны со своими базовыми классами специальным атрибутом `__bases__` — кортежем базовых классов. `__bases__` носит только информационный характер. Фактическая реализация наследования на стадии выполнения использует атрибут `__mro__`. Он содержит кортеж всех родительских классов, перечисленных в порядке поиска. Эта структура — базовая для всех операций, выполняющих чтение, запись или удаление атрибутов экземпляров.

Каждый раз, когда атрибут задается командой `obj.name = value`, вызывается специальный метод `obj.__setattr__('name', value)`. При удалении атрибута командой `del obj.name` вызывается специальный метод `obj.__delattr__('name')`. Поведение этих методов по умолчанию изменяет или удаляет значения из локального словаря `__dict__` объекта `obj`, если только указанный атрибут не соответствует свойству или дескриптору. Тогда запись и удаление будут выполняться функциями записи и удаления, связанными со свойством.

Для обращений к атрибутам вида `obj.name` вызывается специальный метод `obj.__getattr__('name')`. Он проводит поиск атрибута. Это обычно включает проверку свойств, обращение к локальному словарю `__dict__`, проверку словаря класса и поиск в списке MRO. При неудачном

завершении поиска в последней попытке найти атрибут вызывается метод `obj.__getattr__('name')` класса (если он определен). Если и эта попытка оказывается безуспешной, выдается исключение `AttributeError`.

Определяемые пользователями классы при желании могут реализовать свои версии функций обращения к атрибутам. Следующий класс ограничивает имена атрибутов, для которых возможна запись:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __setattr__(self, name, value):
        if name not in {'owner', 'balance'}:
            raise AttributeError(f'No attribute {name}')
        super().__setattr__(name, value)

# Пример
a = Account('Guido', 1000.0)
a.balance = 940.25          # Ok
a.amount = 540.2           # AttributeError. Атрибут amount не найден.
```

Класс, переопределяющий эти методы, должен использовать реализацию по умолчанию, предоставляемую `super()`, для выполнения фактических операций с атрибутами. Дело в том, что реализация по умолчанию берет на себя некоторые базовые аспекты классов — дескрипторы и свойства. Если вы не используете `super()`, вам придется позаботиться об этих нюансах самостоятельно.

7.26. ПРОКСИ, ОБЕРТКИ И ДЕЛЕГИРОВАНИЕ

Иногда класс реализует обертку вокруг другого объекта для создания объекта-заместителя. Прокси — это объект, предоставляющий тот же интерфейс, что и другой объект, но по какой-то причине не связанный с исходным объектом наследованием. Этим он отличается от композиции, где создается совершенно новый объект с другими объектами, но при этом обладающий своим уникальным набором методов и атрибутов.

Прокси могут встречаться во многих реальных сценариях. Например, в распределенных вычислениях фактическая реализация объекта может быть на удаленном сервере в облаке. Клиенты, взаимодействующие с ним, могут использовать прокси, который выглядит как объект на сервере, но при

этом незаметно делегирует все свои вызовы методов с помощью сетевых сообщений.

Типичная реализация заместителей основана на методе `__getattr__()`:

```
class A:
    def spam(self):
        print('A.spam')

    def grok(self):
        print('A.grok')

    def yow(self):
        print('A.yow')

class LoggedA:
    def __init__(self):
        self._a = A()

    def __getattr__(self, name):
        print("Accessing", name)
        # Делегирование внутреннему экземпляру A
        return getattr(self._a, name)

# Пример использования
a = LoggedA()
a.spam()      # Выводит "Accessing spam" и "A.spam"
a.yow()      # Выводит "Accessing yow" и "A.yow"
```

Делегирование иногда используется как альтернатива наследованию:

```
class A:
    def spam(self):
        print('A.spam')

    def grok(self):
        print('A.grok')

    def yow(self):
        print('A.yow')

class B:
    def __init__(self):
        self._a = A()

    def grok(self):
        print('B.grok')
```

```

def __getattr__(self, name):
    return getattr(self._a, name)

# Пример использования
b = B()
b.spam()      # -> A.spam
b.grok()      # -> B.grok (переопределенный метод)
b.yow()       # -> A.yow

```

Здесь все выглядит так, словно класс **B** наследует от класса **A** и переопределяет один метод. Это так, но наследование при этом не используется. Вместо этого **B** хранит внутреннюю ссылку на **A**. Некоторые методы **A** могут переопределяться, но все остальные делегируются методом `__getattr__()`.

Перенаправление поиска атрибутов через `__getattr__()` — стандартный прием. Но учтите, что это не относится к операциям, сопоставленным со специальными методами. Для примера возьмем следующий класс:

```

class ListLike:
    def __init__(self):
        self._items = list()

    def __getattr__(self, name):
        return getattr(self._items, name)

# Пример
a = ListLike()
a.append(1)      # Работает
a.insert(0, 2)   # Работает
a.sort()         # Работает
len(a)           # Не работает, нет метода __len__()
a[0]             # Не работает, нет метода __getitem__()

```

Здесь класс успешно перенаправляет все стандартные методы списков (`list.sort()`, `list.append()` и т. д.) внутреннему списку. Но никакие стандартные операторы Python при этом не работают. Чтобы они заработали, нужно явно реализовать необходимые специальные методы:

```

class ListLike:
    def __init__(self):
        self._items = list()

    def __getattr__(self, name):
        return getattr(self._items, name)

    def __len__(self):
        return len(self._items)

```

```
def __getitem__(self, index):
    return self._items[index]

def __setitem__(self, index, value):
    self._items[index] = value
```

7.27. СОКРАЩЕНИЕ ЗАТРАТ ПАМЯТИ И `__SLOTS__`

Как вы уже знаете, экземпляры хранят свои данные в словаре. Создание большого числа экземпляров может привести к значительным затратам памяти. Если имена атрибутов фиксированы, их можно задать в специальной переменной класса `__slots__`:

```
class Account(object):
    __slots__ = ('owner', 'balance')
    ...
```

`__slots__` может рассматриваться как аннотация определения, позволяющая Python провести оптимизации как по затратам памяти, так и по скорости выполнения. Экземпляры класса со `__slots__` не используют словарь для хранения данных экземпляров. Вместо этого применяется гораздо более компактная структура данных на базе массива. В программах, создающих множество объектов, использование `__slots__` поможет сократить затраты памяти и улучшить скорость выполнения.

Элементами `__slots__` могут быть только атрибуты экземпляров. В нем нет методов, свойств, переменных и других атрибутов уровня класса. Это те же имена, которые обычно являются ключами словаря `__dict__` экземпляра.

Учтите, что `__slots__` сложно взаимодействует с наследованием. Если класс унаследован от базового класса, использующего `__slots__`, то для сохранения преимуществ `__slots__` он должен определить `__slots__` для хранения собственных атрибутов (даже если он их не добавляет). Если вы забудете это сделать, производный класс будет работать медленнее и даже расходовать больше памяти, чем если бы переменная `__slots__` не использовалась ни в одном из классов!

`__slots__` несовместима с множественным наследованием. Если для класса указано несколько базовых классов, каждый из которых содержит непустую переменную `__slots__`, вы получите ошибку `TypeError`.

Использование `__slots__` может нарушить работу кода, предполагающего, что экземпляры содержат атрибут `__dict__`. Хотя к пользовательскому коду это

обычно не относится, служебные библиотеки и другие средства поддержки объектов могут быть запрограммированы для работы с `__dict__` при отладке, сериализации объектов и других операциях.

Присутствие `__slots__` не влияет на вызовы таких методов, как `__getattribute__()`, `__getattr__()` и `__setattr__()`, если они переопределяются в классе. Но при реализации подобных методов учтите, что атрибута `__dict__` экземпляра больше нет.

7.28. ДЕСКРИПТОРЫ

Обычно обращения к атрибутам соответствуют операциям со словарем. Для большего контроля доступ к атрибутам можно направить с помощью определяемых пользователем функций `get`, `set` и `delete`. Использование свойств уже было описано ранее. Но в фактической реализации свойств применяется низкоуровневая конструкция — дескриптор. Это объект уровня класса, управляющий доступом к атрибуту. Реализуя один или несколько из специальных методов `__get__()`, `__set__()` и `__delete__()`, вы можете подключиться напрямую к механизму обращения к атрибутам и адаптировать эти операции для своих целей:

```
class Typed:
    expected_type = object

    def __set_name__(self, cls, name):
        self.key = name

    def __get__(self, instance, cls):
        if instance:
            return instance.__dict__[self.key]
        else:
            return self

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(f'Expected {self.expected_type}')
        instance.__dict__[self.key] = value

    def __delete__(self, instance):
        raise AttributeError("Can't delete attribute")

class Integer(Typed):
    expected_type = int
```

```

class Float(Typed):
    expected_type = float

class String(Typed):
    expected_type = str

# Пример использования:
class Account:
    owner = String()
    balance = Float()

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

```

Здесь класс `Typed` определяет дескриптор, в котором при присваивании атрибуту выполняется проверка типа, а при попытке удаления выдается ошибка. Субклассы `Integer`, `Float` и `String` специализируют `Typed` для конкретного типа. При использовании этих классов в другом классе (например, `Account`) эти атрибуты автоматически вызывают соответствующие методы `__get__()`, `__set__()` или `__delete__()` для обращения:

```

a = Account('Guido', 1000.0)
b = a.owner      # Вызывает Account.owner.__get__(a, Account)
a.owner = 'Eva'  # Вызывает Account.owner.__set__(a, 'Eva')
del f.owner      # Вызывает Account.owner.__delete__(a)

```

Дескрипторы могут быть созданы только на уровне класса. Недопустимо создавать их для каждого экземпляра путем создания объектов дескриптора внутри `__init__()` и других методов. Метод `__set_name__()` дескриптора вызывается после определения класса, но до создания каких-либо экземпляров, чтобы проинформировать дескриптор об имени, использованном внутри класса. Например, определение `balance = Float()` вызывает `Float.__set_name__(Account, 'balance')` для передачи дескриптору информации о классе и используемом имени.

Дескрипторы с методом `__set__()` всегда имеют приоритет над элементами в экземплярном словаре. Например, если имя дескриптора совпадает с именем ключа в словаре экземпляра, предпочтение отдается первому. В примере `Account` выше мы видим, что дескриптор применяет проверку типа, даже при том что словарь экземпляра содержит соответствующий элемент:

```

>>> a = Account('Guido', 1000.0)
>>> a.__dict__
{'owner': 'Guido', 'balance': 1000.0 }

```

```
>>> a.balance = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 63, in __set__
    raise TypeError(f'Expected {self.expected_type}')
TypeError: Expected <class 'float'>
>>>
```

Метод `__get__(instance, cls)` дескриптора получает в аргументах как экземпляр, так и класс. Может оказаться, что `__get__()` будет вызван на уровне класса. Тогда аргумент экземпляра (`instance`) будет содержать `None`. В большинстве случаев `__get__()` возвращает дескриптор, если экземпляр не задан:

```
>>> Account.balance
<__main__.Float object at 0x110606710>
>>>
```

Дескриптор, реализующий только `__get__()`, называется дескриптором метода. Он имеет более слабую привязку, чем дескриптор с возможностями чтения/записи. `__get__()` дескриптора метода вызывается, только если в словаре экземпляра нет соответствующего метода. Он называется дескриптором метода, потому что они в основном используются для реализации разных типов методов Python, включая методы экземпляров, классов и статические методы.

Ниже приведена заготовка реализации, показывающая, как `@classmethod` и `@staticmethod` могли бы быть реализованы с нуля (настоящая реализация более эффективна):

```
import types
class classmethod:
    def __init__(self, func):
        self.__func__ = func

    # Возвращает связанный метод с первым аргументом cls
    def __get__(self, instance, cls):
        return types.MethodType(self.__func__, cls)

class staticmethod:
    def __init__(self, func):
        self.__func__ = func

    # Возвращает минимальную функцию
    def __get__(self, instance, cls):
        return self.__func__
```

Так как дескрипторы методов работают только при отсутствии соответствующего элемента в словаре, они могут использоваться и для реализации разных форм отложенного вычисления атрибутов:

```
class Lazy:
    def __init__(self, func):
        self.func = func

    def __set_name__(self, cls, name):
        self.key = name

    def __get__(self, instance, cls):
        if instance:
            value = self.func(instance)
            instance.__dict__[self.key] = value
            return value
        else:
            return self

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    area = Lazy(lambda self: self.width * self.height)
    perimeter = Lazy(lambda self: 2*self.width + 2*self.height)
```

Здесь атрибуты `area` и `perimeter` вычисляются по требованию и сохраняются в словаре экземпляра. После вычисления значений они возвращаются из словаря экземпляра.

```
>>> r = Rectangle(3, 4)
>>> r.__dict__
{'width': 3, 'height': 4 }
>>> r.area
12
>>> r.perimeter
14
>>> r.__dict__
{'width': 3, 'height': 4, 'area': 12, 'perimeter': 14 }
>>>
```

7.29. ПРОЦЕСС ОПРЕДЕЛЕНИЯ КЛАССА

Определение класса — динамический процесс. При определении класса командой `class` создается новый словарь, который служит локальным

пространством имен класса. Затем тело класса выполняется как сценарий внутри этого пространства. Со временем пространство имен становится атрибутом `__dict__` полученного объекта класса.

В теле класса разрешены любые действительные команды Python. Обычно вы определяете только функции и переменные, но допустимы и управляющие команды, команды импорта, вложенные классы и все остальное. Например, следующий класс содержит условные определения методов:

```
debug = True

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    if debug:
        import logging
        log = logging.getLogger(f'{{__module__}}.{{__qualname__}}')
        def deposit(self, amount):
            Account.log.debug('Depositing %f', amount)
            self.balance += amount

        def withdraw(self, amount):
            Account.log.debug('Withdrawing %f', amount)
            self.balance -= amount
    else:
        def deposit(self, amount):
            self.balance += amount

        def withdraw(self, amount):
            self.balance -= amount
```

В этом примере глобальная переменная `debug` используется для условного определения методов. `__qualname__` и `__module__` содержат заранее определенные строки с информацией об имени класса и содержащем модуле. Она может использоваться командами в теле класса. В этом примере они предназначены для настройки конфигурации системы ведения журнала. Возможно, есть и более элегантные способы структурирования этого кода, но ключевой момент в том, что в класс можно поместить что угодно.

Важно: пространство имен, используемое для хранения содержимого тела класса, не является областью видимости переменных. Любое имя, которое используется в методе (такое как `Account.log` в прошлом примере), должно быть полностью уточненным.

Если функция `locals()` используется в теле класса (но не внутри метода), то она возвращает словарь, используемый для пространства имен класса.

7.30. ДИНАМИЧЕСКОЕ СОЗДАНИЕ КЛАССА

Обычно классы создаются командой `class`, но необязательно. Как упоминалось ранее, классы определяются выполнением тела класса для заполнения пространства имен. Если вы можете заполнить словарь своими определениями, класс может быть создан и без команды `class`. Для этого используется конструкция `types.new_class()`:

```
import types

# Методы (не содержащиеся в классе)
def __init__(self, owner, balance):
    self.owner = owner
    self.balance = balance

def deposit(self, amount):
    self.balance += amount

def withdraw(self, amount):
    self.balance -= amount

methods = {
    '__init__': __init__,
    'deposit': deposit,
    'withdraw': withdraw,
}

Account = types.new_class('Account', (),
                           exec_body=lambda ns: ns.update(methods))

# Класс создан
a = Account('Guido', 1000.0)
a.deposit(50)
a.withdraw(25)
```

`new_class()` должна получать имя класса, кортеж базовых классов и функцию обратного вызова, ответственную за заполнение пространства имен классов. Функция обратного вызова получает в аргументе словарь пространства имен класса. Этот словарь должен обновляться на месте. Возвращаемое значение функции обратного вызова игнорируется.

Динамическое создание классов может быть полезным при построении классов на основе структур данных. Например, в разделе, посвященном дескрипторам, были определены следующие классы:

```
class Integer(Typed):
    expected_type = int

class Float(Typed):
```

```

    expected_type = float

class String(Typed):
    expected_type = str

```

Код получается в высшей степени однообразным. Пожалуй, подход, основанный на данных, здесь подойдет лучше:

```

typed_classes = [
    ('Integer', int),
    ('Float', float),
    ('String', str),
    ('Bool', bool),
    ('Tuple', tuple),
]

globals().update(
    (name, types.new_class(name, (Typed,),
        exec_body=lambda ns: ns.update(expected_type=ty)))
    for name, ty in typed_classes)

```

В этом примере глобальное пространство имен модуля обновляется классами, динамически создаваемыми с помощью `types.new_class()`. Чтобы создать больше классов, поместите соответствующий элемент в список `typed_classes`.

Иногда для динамического создания классов используется функция `type()`:

```
Account = type('Account', (), methods)
```

Такое решение работает, но оно не учитывает некоторые высокоуровневые механизмы классов, например метаклассы (о которых будет рассказано ниже). В современном коде лучше использовать `types.new_class()`.

7.31. МЕТАКЛАССЫ

При определении класса в Python само определение класса становится объектом:

```

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

```

```

def withdraw(self, amount):
    self.balance -= amount

isinstance(Account, object)    # -> True

```

Присмотревшись, можно понять, что если `Account` — это объект, то его что-то должно создать. Созданием объекта класса управляет его особая разновидность — метакласс. Это класс, создающий экземпляры классов.

В прошлом примере метаклассом, который создал `Account`, был встроенный класс `type`. При проверке типа `Account` вы увидите, что класс — экземпляр `type`:

```

>>> Account.__class__
<type 'type'>
>>>

```

Сначала ничего не понятно, но происходит примерно то же, что и с целыми числами. Например, если написать `x = 42` и проверить `x.__class__`, вы получите `int` — класс, создающий целые числа. Точно так же `type` создает экземпляры типов или классов.

При определении нового класса командой `class` происходит ряд событий. Сначала создается новое пространство имен для класса. Затем тело класса выполняется в этом пространстве. Наконец, имя класса, базовые классы и заполненное пространство имен используются для создания экземпляра класса. Следующий код показывает выполняемые низкоуровневые операции:

```

# Шаг 1: Создание пространства имен класса
namespace = type.__prepare__('Account', ())

# Шаг 2: Выполнение тела класса
exec('''
def __init__(self, owner, balance):
    self.owner = owner
    self.balance = balance

def deposit(self, amount):
    self.balance += amount

def withdraw(self, amount):
    self.balance -= amount
''', globals(), namespace)

# Шаг 3: Создание итогового объекта класса
Account = type('Account', (), namespace)

```

В процессе определения существует взаимодействие с классом `type` для создания пространства имен класса и его итогового объекта. Вы можете выбрать не только `type`. Есть вариант обработки класса другим классом типа с указанием другого метакласса. Для этого при наследовании указывается другой ключевой аргумент `metaclass`:

```
class Account(metaclass=type):
    ...
```

Если аргумент `metaclass` не задан, команда `class` проверяет тип первого элемента в кортеже базовых классов (если он есть) и использует его как метакласс. Если вы используете запись `class Account(object)`, полученный класс `Account` будет иметь такой же тип, как и объект (то есть `type`). Обратите внимание: классы, для которых родитель вообще не задан, всегда унаследованы от `object`, поэтому такой принцип все равно работает.

Для создания нового метакласса определите класс, унаследованный от `type`. В нем можно переопределить один или несколько методов, используемых при создании класса. Обычно к их числу относится метод `__prepare__()`, используемый для создания пространства имен класса; `__new__()`, используемый для создания экземпляра класса; `__init__()`, вызываемый после того, как класс уже будет создан; и `__call__()`, используемый для создания новых экземпляров.

В следующем примере реализуется метакласс, который просто выводит входные аргументы каждого метода, чтобы вы могли поэкспериментировать с ними:

```
class mytype(type):

    # Создание пространства имен класса
    @classmethod
    def __prepare__(meta, clsname, bases):
        print("Preparing:", clsname, bases)
        return super().__prepare__(clsname, bases)

    # Создание экземпляра класса после выполнения тела
    @staticmethod
    def __new__(meta, clsname, bases, namespace):
        print("Creating:", clsname, bases, namespace)
        return super().__new__(meta, clsname, bases, namespace)

    # Инициализация экземпляра класса
    def __init__(cls, clsname, bases, namespace):
        print("Initializing:", clsname, bases, namespace)
        super().__init__(clsname, bases, namespace)
```

```

# Создание новых экземпляров класса
def __call__(cls, *args, **kwargs):
    print("Creating instance:", args, kwargs)
    return super().__call__(*args, **kwargs)

# Пример
class Base(metaclass=mytype):
    pass

# Определение Base генерирует следующий вывод
# Preparing: Base ()
# Creating: Base () {'__module__': '__main__', '__qualname__': 'Base'}
# Initializing: Base () {'__module__': '__main__', '__qualname__':
'Base'}

b = Base()
# Creating instance: () {}

```

Один из простейших аспектов использования метаклассов — присваивание имен переменным и отслеживание сущностей, задействованных в процессе. В приведенном коде имя `meta` относится к самому метаклассу, `cls` — к экземпляру класса, созданному метаклассом, а `self`, хотя здесь оно и не используется, относится к обычному экземпляру, создаваемому классом.

Метаклассы распространяются при наследовании. Поэтому если вы определили базовый класс, использующий другой метакласс, все дочерние классы тоже будут его использовать. Следующий пример поможет увидеть ваш нестандартный метакласс в действии:

```

class Account(Base):
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

print(type(Account)) # -> <class 'mytype'>

```

Метаклассы чаще всего используются, когда вы хотите поддерживать крайне низкоуровневый контроль над средой определения классов и процессом создания. Но прежде чем продолжать, напомним, что Python уже дает большой объем функциональности для наблюдения и изменения определений классов (например, метод `__init_subclass__()`, декораторы классов, дескрипторы,

примеси и т. д.). В большинстве случаев метаклассы вам не понадобятся. Но в нескольких примерах будут представлены ситуации, где метакласс будет единственным разумным решением.

Одно из возможных применений метакласса — перезапись содержимого пространства имен классов до создания объекта класса. Некоторые возможности классов устанавливаются в момент определения и не могут изменяться позднее. Одна из таких — специальная переменная `__slots__`. Как упоминалось ранее, `__slots__` обеспечивает оптимизацию производительности, связанную с затратами памяти экземпляров. Метакласс из примера ниже автоматически задает атрибут `__slots__` на основании сигнатуры вызова метода `__init__()`.

```
import inspect

class SlotMeta(type):
    @staticmethod
    def __new__(meta, clsname, bases, methods):
        if '__init__' in methods:
            sig = inspect.signature(methods['__init__'])
            __slots__ = tuple(sig.parameters)[1:]
        else:
            __slots__ = ()
        methods['__slots__'] = __slots__
        return super().__new__(meta, clsname, bases, methods)

class Base(metaclass=SlotMeta):
    pass

# Пример
class Point(Base):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Здесь `Point` автоматически создается с переменной `__slots__`, содержащей ('x', 'y'). Полученные экземпляры `Point` теперь получают экономию памяти, не зная, что используются `__slots__`. Ее не нужно задавать напрямую. Такие трюки невозможны с декораторами классов или с `__init_subclass__()` — эти средства работают с классом только после его создания. К тому времени применять оптимизацию `__slots__` уже поздно.

Другое применение метаклассов — изменение окружения определения класса. Например, дублирующиеся определения имен при определении класса обычно приводят к незаметной ошибке, где второе определение замещает первое. Допустим, вы хотите обнаруживать такую ситуацию. Следующий

метакласс делает это, определяя специальную разновидность словаря для пространства имен классов:

```
class NoDupeDict(dict):
    def __setitem__(self, key, value):
        if key in self:
            raise AttributeError(f'{key} already defined')
        super().__setitem__(key, value)

class NoDupeMeta(type):
    @classmethod
    def __prepare__(meta, clsname, bases):
        return NoDupeDict()

class Base(metaclass=NoDupeMeta):
    pass

# Пример
class SomeClass(Base):
    def yow(self):
        print('Yow!')

    def yow(self, x): # Ошибка. Имя уже определено
        print('Different Yow!')
```

Это только небольшой пример доступных возможностей. Разработчику фреймворков метаклассы дают возможность жесткого контроля за происходящим при определении класса. Это позволяет использовать классы как язык, специфический для предметной области.

Традиционно метаклассы использовались для решения многочисленных задач, которые теперь могут решаться иначе. В частности, метод `__init_subclass__()` может использоваться для разных сценариев, где когда-то применялись метаклассы. К их числу относится регистрация классов в центральном реестре, автоматическое декорирование методов и генерация кода.

7.32. ВСТРОЕННЫЕ ОБЪЕКТЫ ДЛЯ ЭКЗЕМПЛЯРОВ И КЛАССОВ

В этом разделе вы узнаете о низкоуровневых объектах для представления типов и экземпляров. Эта информация может пригодиться при низкоуровневом метапрограммировании и написании кода для прямых манипуляций с типами.

В табл. 7.1 приведены часто используемые атрибуты объекта типа `cls`.

Таблица 7.1. Атрибуты типа

| АТРИБУТ | ОПИСАНИЕ |
|--------------------------------------|--|
| <code>cls.__name__</code> | Имя класса |
| <code>cls.__module__</code> | Имя модуля, где определяется класс |
| <code>cls.__qualname__</code> | Полностью уточненное имя класса |
| <code>cls.__bases__</code> | Кортеж базовых классов |
| <code>cls.__mro__</code> | Кортеж порядка разрешения методов (MRO) |
| <code>cls.__dict__</code> | Словарь с переменными и методами класса |
| <code>cls.__doc__</code> | Строка документации |
| <code>cls.__annotations__</code> | Словарь аннотаций типов класса |
| <code>cls.__abstractmethods__</code> | Множество имен абстрактных методов (может быть не определено, если их нет) |

Атрибут `cls.__name__` содержит короткое имя класса. `cls.__qualname__` содержит полностью уточненное имя с дополнительной информацией о контексте среды (может быть полезно, если класс определяется внутри функции или если вы создаете вложенное определение класса). Словарь `cls.__annotations__` содержит аннотации типов уровня класса (если они есть).

В табл. 7.2 приведены специальные атрибуты экземпляра `i`.

Таблица 7.2. Атрибуты экземпляра

| АТРИБУТ | ОПИСАНИЕ |
|--------------------------|---|
| <code>i.__class__</code> | Класс, к которому принадлежит экземпляр |
| <code>i.__dict__</code> | Словарь с данными экземпляра (если определен) |

Атрибут `__dict__` содержит словарь, где хранятся все данные, связанные с экземпляром. Но если определенный пользователем класс использует `__slots__`, применяется более эффективное внутреннее представление, и экземпляры не содержат атрибут `__dict__`.

7.33. НАПОСЛЕДОК: БУДЬТЕ ПРОЩЕ

В этой главе много информации о классах и возможностях настройки и управления ими. Но лучшей стратегией при написании классов часто оказывается максимальная простота. Да, вы можете использовать абстрактные

базовые классы, метаклассы, дескрипторы, декораторы классов, свойства, множественное наследование, примеси, паттерны и аннотации типов. Но вы можете написать и обычный класс. Скорее всего, его будет достаточно и все поймут, что он делает.

Размышляя над общей картиной, полезно остановиться и рассмотреть некоторые желательные качества кода. Во-первых, очень важна удобочитаемость — а она часто страдает при нагромождениях нескольких уровней абстракции. Во-вторых, код должен быть по возможности простым для наблюдения и отладки, и не забывайте об использовании REPL. Наконец, возможность тестировать код — часто отличный стимул для хорошей структуры. Если ваш код невозможно протестировать или его тестирование связано с большими неудобствами, возможно, стоит поискать более эффективную организацию вашего решения.

ГЛАВА 8

Модули и пакеты

Программы Python делятся на модули и пакеты, загружаемые командой `import`. В этой главе система модулей и пакетов описывается подробно. Основное внимание уделяется программированию с модулями и пакетами, а не процессу упаковки кода для развертывания на других компьютерах. За этой информацией обращайтесь к новейшей документации по адресу <https://packaging.python.org/tutorials/packaging-projects/>.

8.1. МОДУЛИ И КОМАНДА `IMPORT`

Любой файл с исходным кодом Python может быть импортирован как модуль. Пример:

```
# module.py
a = 37

def func():
    print(f'func says that a is {a}')

class SomeClass:
    def method(self):
        print('method says hi')

print('loaded module')
```

Этот файл содержит общие программные элементы, включая глобальную переменную, функцию, определение класса и изолированную команду. На примере показаны некоторые важные (и иногда неочевидные) аспекты загрузки модулей.

Для загрузки модуля используйте команду `import`:

```
>>> import module
loaded module
```

```
>>> module.a
37
>>> module.func()
func says that a is 37
>>> s = module.SomeClass()
>>> s.method()
method says hi
>>>
```

При выполнении `import` происходят следующие операции.

1. Программа ищет исходный код модуля. Если его нет, выдается исключение `ImportError`.
2. Создается новый объект модуля. Он служит контейнером для всех глобальных определений в модуле. Иногда он называется *пространством имен*.
3. Исходный код модуля выполняется в только что созданном пространстве имен модуля.
4. Если выполнение проходит без ошибок, на стороне вызова создается имя, которое ссылается на новый объект модуля. Оно совпадает с именем модуля, но не имеет суффикса имени файла. Например, если код хранится в файле `module.py`, то модулю будет присвоено имя `module`.

Из этих шагов сложнее всего первый (поиск модулей). Неопытные программисты часто сталкиваются с ошибками из-за использования неподходящего имени файла или размещения кода в неизвестном месте. Имя файла модуля должно подчиняться тем же правилам, что и имена переменных (буквы, цифры и подчеркивания), и иметь суффикс `.py`, например `module.py`. При использовании `import` имя указывается без суффикса: `import module`, а не `import module.py` (в последнем случае будет выведено непонятное сообщение об ошибке). Файл должен быть в одном из каталогов, перечисленных в `sys.path`.

Остальные шаги связаны с изолированной средой, определяемой модулем для кода. Все определения, присутствующие в модуле, остаются изолированными в нем. Так мы избежим конфликта переменных, функций и классов с идентичными именами в других модулях. При обращении к определениям в модуле используются полностью уточненные имена вида `module.func()`.

`import` выполняет все команды в загруженном исходном файле. Если модуль проводит вычисления или генерирует вывод в дополнение к определению объектов, вы увидите результат — вывод сообщения `loaded module` в приведенном примере. Типичная ошибка использования модулей связана с обращением

к классам. Модуль всегда определяет пространство имен, и если в файле `module.py` определяется класс `SomeClass`, для ссылок на него нужно использовать имя `module.SomeClass`.

Для импорта нескольких модулей одной командой `import` укажите список имен, разделенных запятыми:

```
import socket, os, re
```

Иногда локальное имя, используемое для обращения к модулю, изменяется квалификатором `as` в команде `import`:

```
import module as mo
mo.func()
```

Второй стиль импорта считается стандартной практикой при анализе данных. Часто встречаются следующие команды:

```
import numpy as np
import pandas as pd
import matplotlib as plt
...
```

При переименовании модуля новое имя применяется только в контексте выполнения команды `import`. Другие независимые модули все еще могут загружать модуль по его исходному имени.

Присваивание другого имени импортируемому модулю может стать полезным инструментом для управления разными реализациями стандартной функциональности или для написания расширяемых программ. Если у вас есть два модуля `unixmodule.py` и `winmodule.py`, оба определяющие функцию `func()`, но использующие платформенно-независимые подробности реализации, вы можете написать код для избирательного импортирования модуля:

```
if platform == 'unix':
    import unixmodule as module
elif platform == 'windows':
    import winmodule as module

...
r = module.func()
```

В Python модули — это первоклассные объекты. Они могут присваиваться переменным, сохраняться в структурах данных и передаваться в программе как данные. Например, имя `module` в приведенном примере представляет переменную, обозначающую соответствующий объект модуля.

8.2. КЕШИРОВАНИЕ МОДУЛЕЙ

Исходный код модуля загружается и выполняется только один раз, независимо от числа раз использования `import`. Последующие команды `import` связывают имя модуля с его объектом, уже созданным прошлой командой.

Многие новички импортируют модуль в интерактивный сеанс и изменяют исходный код (например, для исправления ошибки), но новая команда `import` не загружает измененный код. Проблема возникает из-за кеширования модулей. Python никогда не перезагружает ранее импортированный модуль, если его исходный код был изменен.

Кеш всех модулей, загруженных на данный момент, доступен в `sys.modules` — словаре, связывающем имена модулей с их объектами. Содержимое словаря определяет, загружает `import` свежую копию модуля или нет. При удалении модуля из кеша он будет снова загружен следующей командой `import`. Обычно это небезопасно по причинам, описанным в разделе 8.5.

Иногда команда `import` используется в функциях, как в следующем примере:

```
def f(x):  
    import math  
    return math.sin(x) + math.cos(x)
```

Кажется, что такая реализация будет невероятно медленной — модуль загружается при каждом вызове. На самом деле затраты на импортирование будут минимальными: все сводится к одному поиску по словарю, ведь Python немедленно находит модуль в кеше. Основные возражения против импортирования из функций связаны со стилем программирования.

Обычно все команды импортирования модулей перечисляются в начале файла, где их удобнее просматривать. С другой стороны, если у вас есть специализированная функция, которая редко вызывается, размещение зависимостей импортирования функции внутри тела может ускорить загрузку программы. В этом случае важные модули будут загружаться, только если они действительно нужны.

8.3. ИМПОРТИРОВАНИЕ ОТДЕЛЬНЫХ ИМЕН ИЗ МОДУЛЯ

Вы можете загрузить отдельные определения из модуля в текущее пространство имен командой `from модуль import имя`. Команда идентична `import`, только вместо создания имени, ссылающегося на созданное пространство

имен модуля, она размещает в текущем пространстве имен ссылки на один или несколько объектов, определенных в модуле:

```
from module import func # Импортирует модуль и включает
                        # func в текущее пространство имен
func()                  # Вызывает функцию func(), определенную в модуле
module.func()           # Ошибка. NameError: module
```

Если вы хотите импортировать сразу несколько определений, команде `from` можно передать несколько имен, разделенных запятыми:

```
from module import func, SomeClass
```

Семантически команда `from модуль import имя` копирует имя из кеша модуля в локальное пространство имен. Иначе говоря, Python сначала незаметно выполняет `import module`, а после выполняется присваивание из кеша локальному имени, например `name = sys.modules['module'].name`.

Есть распространенное заблуждение, что команда `from модуль import имя` более эффективна, так как может загружать только часть модуля. Это не так. В любом случае весь модуль загружается и сохраняется в кеше.

Импортирование функций в синтаксисе `from` не изменяет их правил области видимости. Когда функции ищут переменные, они проверяют только файл, где функция была определена, но не пространство имен, в котором эта функция импортируется и вызывается:

```
>>> from module import func
>>> a = 42
>>> func()
func says that a is 37
>>> func.__module__
'module'
>>> func.__globals__['a']
37
>>>
```

Похожие заблуждения относятся и к поведению глобальных переменных. Рассмотрим код, импортирующий `func` и глобальную переменную `a`:

```
from module import a, func
a = 42          # Изменяет переменные
func()          # Выводит "func says a is 37"
print(a)        # Выводит "42"
```

Присваивание переменной в Python не является операцией сохранения в памяти. Иначе говоря, имя `a` в этом примере не представляет ячейку памяти,

где сохраняется значение. Исходная команда `import` связывает локальное имя `a` с исходным объектом `module.a`. Но следующее повторное присваивание `a = 42` переключает локальное имя `a` на совершенно другой объект. В этой точке переменная `a` уже не связана со значением в импортированном модуле. Из-за этого команда `from` не может использоваться так, чтобы переменные вели себя как глобальные в языках вроде С. Чтобы использовать изменяемые глобальные параметры в своей программе, поместите их в модуль и явно используйте имя модуля командой `import`, например `module.a`.

Универсальный символ `*` (звездочка) иногда используется для загрузки всех определений в модуле, кроме тех, имена которых начинаются с подчеркивания:

```
# Загрузить все определения в текущее пространство имен
from module import *
```

Команда `from модуль import *` может использоваться только в области видимости верхнего уровня модуля. Эта форма `import` не может использоваться в теле функции. Модули могут точно управлять набором имен, импортируемых командой `from модуль import *`, для чего определяют список `__all__`:

```
# module: module.py
__all__ = [ 'func', 'SomeClass' ]
a = 37          # Не экспортируется
def func():     # Экспортируется
    ...
class SomeClass: # Экспортируется
    ...
```

В интерактивном приглашении Python команда `from модуль import *` предоставляет удобные средства работы с модулем. Но лучше не использовать этот стиль `import` в программах. Злоупотребления приводят к загрязнению локального пространства имен и порождают путаницу:

```
from math import *
from random import *
from statistics import *

a = gauss(1.0, 0.25) # Из какого модуля?
```

Лучше явно указывать имена:

```
from math import sin, cos, sqrt
from random import gauss
from statistics import mean

a = gauss(1.0, 0.25)
```

8.4. ЦИКЛИЧЕСКИЙ ИМПОРТ

Специфическая проблема возникает, когда два модуля импортируют друг друга. Допустим, у вас есть два файла:

```
# -----
# moda.py

import modb

def func_a():
    modb.func_b()

class Base:
    pass
# -----
# modb.py

import moda

def func_b():
    print('B')

class Child(moda.Base):
    pass
```

В этом коде возникает странная зависимость порядка импортирования. Если сначала выполняется команда `import modb`, все нормально, но если поставить на первое место `import moda`, происходит ошибка с сообщением о том, что значение `moda.Base` не определено.

Для лучшего понимания ситуации нужно проанализировать последовательность выполнения. `import moda` начинает выполнять файл `moda.py`. Сначала применяется команда `import modb`, а после управление передается `modb.py`. Файл открывается командой `import moda`. Эта команда не входит в рекурсивный цикл, а выполняется из кеша модулей.

Выполнение продолжается со следующей команды в `modb.py`. Это хорошо: циклический импорт не приводит к зависанию Python или переходу в новое пространственно-временное измерение. Но в этой точке выполнения модуль `moda` был вычислен только частично. После передачи управления команде `class Child(moda.Base)` происходит фатальный сбой. Необходимый класс `Base` еще не был определен.

Возможное решение проблемы — перемещение команды `import modb` в другую точку. Ее можно переместить в функцию `func_a()`, где это определение фактически используется:


```
# moda.py

def func_a():
    import modb
    modb.func_b()

class Base:
    pass
```

Также можно переместить `import` ближе к концу файла:

```
# moda.py

def func_a():
    modb.func_b()

class Base:
    pass

import modb      # Команда должна следовать после определения Base
```

Скорее всего, оба этих решения вызовут недоумение на рецензировании кода. Обычно команды импорта модулей не размещаются в конце файла. Циклический импорт почти всегда указывает на проблемы со структурой кода. В таких случаях лучше переместить определение `Base` в отдельный файл `base.py` и переписать `modb.py` так:

```
# modb.py

import base

def func_b():
    print('B')

class Child(base.Base):
    pass
```

8.5. ПЕРЕЗАГРУЗКА И ВЫГРУЗКА МОДУЛЕЙ

Надежной поддержки перезагрузки или выгрузки ранее импортированных модулей нет. Вы можете удалить модуль из `sys.modules`, но это не приведет к выгрузке модуля из памяти.

Дело в том, что ссылки на кешированный объект модуля все еще остаются в других модулях, импортировавших этот. Более того, если в модуле определяются экземпляры классов, они содержат ссылки на свои объекты классов, в которых есть ссылки на модуль, где они были определены.

Из-за того, что ссылки на модули есть во многих местах, перезагрузка модуля после внесения изменений в его реализацию становится невозможной. Например, если вы удалите модуль из `sys.modules` и попытаетесь перезагрузить его командой `import`, это не приведет к изменению всех существующих ссылок на модуль в программе.

Вместо этого есть одна ссылка на новый модуль, созданная самой последней командой `import`, и множество — на старый модуль, созданный командами `import` в других частях кода. А это не то, что вам нужно. Перезагрузка модулей никогда не будет безопасной при реальной эксплуатации, если только вы не можете тщательно контролировать всю исполнительную среду.

Для перезагрузки модулей есть функция `reload()` из библиотеки `importlib`. В аргументе функции передается уже загруженный модуль:

```
>>> import module
>>> import importlib
>>> importlib.reload(module)
loaded module
<module 'module' from 'module.py'>
>>>
```

`reload()` загружает новую версию исходного модуля кода и выполняет ее в уже существующем пространстве имен модуля. Ранее существовавшее пространство имен не очищается. Это то же самое, как когда вы вводите новый исходный код поверх старого без перезапуска интерпретатора.

Если другие модули ранее импортировали перезагруженный модуль с помощью стандартного оператора `import` (`import модуль`), перезагрузка заставит их увидеть обновленный код как по волшебству. Но такое решение сопряжено с высоким риском. Во-первых, функция не перезагружает никакие модули, которые могли быть импортированы перезагруженным файлом. Перезагрузка выполняется только для одного модуля, переданного `reload()`. Во-вторых, если какой-либо модуль использовал форму импорта `from модуль import имя`, эффект перезагрузки не отразится на этом импорте. Наконец, если были созданы экземпляры классов, перезагрузка не изменит нижележащего определения класса.

Теперь в одной программе появляются и два разных определения одного класса — старое, которое продолжает использоваться всеми существующими экземплярами при перезагрузке, и новое, которое используется для новых экземпляров. Такое раздвоение создает путаницу.

Заметьте, что расширения Python, написанные на C/C++, не могут безопасно выгружаться или перезагружаться этим способом. Для этого не предусмотрено никакой поддержки, и базовая операционная система может запретить

такой способ. Лучший выход в подобной ситуации — перезапуск процесса интерпретатора Python.

8.6. КОМПИЛЯЦИЯ МОДУЛЕЙ

При первом импорте модуль компилируется в байт-код интерпретатора, который сохраняется в файле `.pyc` в специальном каталоге `__pycache__`. Этот каталог обычно находится там же, где и исходный файл `.py`. Когда такая же команда импорта выполняется при другом запуске программы, будет загружен откомпилированный байт-код. Это значительно ускоряет процесс.

Кеширование байт-кода — автоматический процесс. При изменении исходного кода файлы автоматически генерируются заново. Все это просто работает без вашего участия.

Но все еще остаются причины знать о процессе кеширования и компиляции. Во-первых, иногда файлы Python устанавливаются (часто непреднамеренно) в среде, где у пользователей нет разрешений операционной системы для создания необходимого каталога `__pycache__`. Python будет работать, но теперь при каждой операции импорта он будет загружать исходный код и компилировать его в байт-код. Программы будут загружаться намного медленнее. Точно так же при развертывании или упаковке приложения Python может быть полезно включить откомпилированный байт-код, ведь это может значительно ускорить запуск программы.

Есть и другая веская причина знать о кешировании модулей: оно используется некоторыми средствами программирования. Базовые приемы метапрограммирования, в том числе использующие динамическую генерацию кода и функцию `exec()`, компенсируют выигрыш от кеширования байт-кода. Яркий пример — использование `@dataclass`:

```
from dataclasses import dataclass
```

```
@dataclass
class Point:
    x: float
    y: float
```

Работа `@dataclass` основана на генерации функций методов в виде текстовых фрагментов и их выполнении функцией `exec()`. Генерируемый код не кешируется системой импорта. С одним определением класса никаких плюсов вы не получите. Но модуль из 100 классов `@dataclass` импортируется приблизительно в 20 раз медленнее аналогичного, где классы записываются традиционно, но менее компактно.

8.7. ПУТЬ ПОИСКА МОДУЛЕЙ

При импорте модулей интерпретатор проводит поиск по списку каталогов из переменной `sys.path`. Первым элементом `sys.path` обычно бывает пустая строка `' '`, обозначающая текущий рабочий каталог. При выполнении сценария первый элемент `sys.path` содержит каталог, где находится сценарий. В других элементах `sys.path` обычно хранятся имена каталогов и архивных файлов с расширениями `.zip`. Порядок перечисления элементов в `sys.path` определяет порядок поиска при импорте модулей. Для включения новых элементов в путь поиска добавьте их в этот список. Это можно сделать напрямую или присваиванием переменной среды `PYTHONPATH`. В UNIX это делается так:

```
bash $ env PYTHONPATH=/some/path python3 script.py
```

Архивные ZIP-файлы дают возможность объединения набора модулей в один файл. Допустим, вы создали два модуля, `foo.py` и `bar.py`, и поместили их в файл `mymodules.zip`. Его можно добавить в путь поиска Python так:

```
import sys
sys.path.append('mymodules.zip')
import foo, bar
```

Для поиска можно указать и отдельные подкаталоги в структуре каталогов `.zip`-файла. `.zip`-файлы могут объединяться и с обычными компонентами путей:

```
sys.path.append('/tmp/modules.zip/lib/python')
```

Использование суффикса `.zip` для ZIP-файлов необязательно. Раньше в пути поиска часто встречались и файлы `.egg`. Файлы `.egg` появились в начальном инструментарии управления пакетами Python — `setuptools`. Но файл `.egg` — это просто обычный файл или каталог в формате ZIP с добавлением метаданных (номер версии, зависимости и т. д.).

8.8. ВЫПОЛНЕНИЕ В КАЧЕСТВЕ ОСНОВНОЙ ПРОГРАММЫ

Хотя этот раздел посвящен команде `import`, файлы Python часто выполняются в качестве основного сценария:

```
% python3 module.py
```

В каждом модуле есть переменная `__name__`, где хранится его имя. Код может проверить эту переменную, чтобы определить, в каком модуле она

выполняется. Модуль верхнего уровня в интерпретаторе называется `__main__`. Программы, заданные в командной строке или введенные в интерактивном режиме, выполняются в `__main__`.

Иногда программа может изменить это поведение в зависимости от того, была ли она импортирована в виде модуля или выполняется в `__main__`. Например, модуль может включать код, который выполняется, если модуль выполняется как основная программа, но игнорируется при простом импорте модуля другим модулем.

```
# Проверить, в каком режиме выполняется модуль
if __name__ == '__main__':
    # Да, выполняется как основной сценарий
    команды
else:
    # Нет, импортируется как модуль
    команды
```

Исходные файлы для использования в качестве библиотек могут применять такие проверки для включения необязательного тестового кода или примеров. При разработке модулей можно поместить отладочный код для тестирования функциональности вашей библиотеки внутри команды `if` и выполнять код Python вашего модуля как основную программу. Этот код не будет выполняться у пользователей, импортирующих вашу библиотеку. Каталог с кодом Python можно выполнить, если он содержит специальный файл `__main__.py`. Допустим, вы создаете такой каталог:

```
myapp/
  foo.py
  bar.py
  __main__.py
```

Вы можете применить этот каталог в Python командой `python3 myapp`. Выполнение начнется с файла `__main__.py`. Этот способ работает, и если вы преобразуете каталог `myapp` в ZIP-архив. Команда `python3 myapp.zip` ищет файл `__main__.py` на верхнем уровне структуры и выполняет его, если он будет найден.

8.9. ПАКЕТЫ

В любых программах, кроме самых простых, код Python упорядочивается в *пакеты*. Пакет — это набор модулей, сгруппированных под общим именем верхнего уровня. Такое упорядочивание помогает разрешить конфликты между именами модулей в разных приложениях и отделить ваш код от

стороннего. Для определения пакета создайте каталог с отличительным именем и поместите в него файл `__init__.py` (в исходном состоянии этот файл пуст). Затем разместите в этом каталоге нужные файлы Python и подпакеты. Пакет может иметь следующую структуру:

```
graphics/
  __init__.py
  primitive/
    __init__.py
    lines.py
    fill.py
    text.py
    ...
  graph2d/
    __init__.py
    plot2d.py
    ...
  graph3d/
    __init__.py
    plot3d.py
    ...
  formats/
    __init__.py
    gif.py
    png.py
    tiff.py
    jpeg.py
```

Команда `import` используется для загрузки модулей из пакетов так же, как и для простых модулей, не считая более длинных имен:

```
# Полный путь
import graphics.primitive.fill
...
graphics.primitive.fill.floodfill(img, x, y, color)

# Загрузка конкретного подмодуля
from graphics.primitive import fill
...
fill.floodfill(img, x, y, color)

# Загрузка конкретной функции из подмодуля
from graphics.primitive.fill import floodfill
...
floodfill(img, x, y, color)
```

Когда какая-то часть пакета импортируется впервые, сначала выполняется код из файла `__init__.py` (если он есть). Как уже упоминалось, этот файл может

быть пустым, но он может и содержать код, выполняющий специфическую инициализацию для конкретного пакета. При импорте глубоко вложенного подмодуля выполняются все файлы `__init__.py`, встречающиеся при обходе структуры каталогов. Так, команда `import graphics.primitive.fill` сначала выполнит файл `__init__.py` из каталога `graphics/`, а затем `__init__.py` из `primitive/`.

Наблюдательные пользователи заметят, что пакет работает даже при отсутствии файлов `__init__.py`. Это правда — каталог с кодом Python можно использовать как пакет даже без файла `__init__.py`. Но здесь есть один неочевидный момент: каталог с отсутствующим файлом `__init__.py` в действительности определяет другую разновидность пакета — *пакет пространства имен*. Это расширенная функциональность, иногда используемая очень большими библиотеками и фреймворками для реализации систем с плагинами. Обычно это нежелательно — лучше всегда добавлять файлы `__init__.py` при создании пакетов.

8.10. ИМПОРТ ИЗ ПАКЕТА

Важнейшая особенность команды `import` в том, что все команды импорта модулей требуют абсолютного или полностью уточненного (полного) пути в пакете. Это относится и к командам `import` в самом пакете. Допустим, `graphics.primitive.fill` хочет импортировать модуль `graphics.primitive.lines`. Простая команда вида `import lines` не сработает — вы получите исключение `ImportError`. Вместо этого нужно полностью уточнить импортируемое имя:

```
# graphics/primitives/fill.py
# Полностью уточненное импортирование подмодуля
from graphics.primitives import lines
```

Запись полных имен пакетов утомительна и ненадежна. Иногда лучше переименовать пакет — например, чтобы использовать его разные версии. Если имя пакета жестко фиксируется в коде, это сделать не удастся. Лучше использовать команду импорта пакета:

```
# graphics/primitives/fill.py
# Импортирование относительно пакета
from . import lines
```

Здесь точка (.) в команде `from . import lines` обозначает каталог с импортирующим модулем. Эта команда ищет модуль `lines` в одном каталоге с файлом `fill.py`.

При относительном импорте можно задать подмодули в разных каталогах одного пакета. Например, если модуль `graphics.graph2d.plot2d` хочет импортировать `graphics.primitive.lines`, он может использовать такую команду:

```
# graphics/graph2d/plot2d.py

from ..primitive import lines
```

Здесь `..` — это перемещение на один уровень вверх, а `primitive` выбирает другой подкаталог подпакета.

Относительный импорт может задаваться только в форме `from модуль import имя` команды `import`. Так, команды `import ..primitive.lines` или `import .lines` содержат синтаксическую ошибку. Кроме того, *имя* должно быть простым идентификатором, поэтому команда вида `from .. import primitive.lines` также недопустима. Наконец, относительный импорт может использоваться только внутри пакета. Применять его для обращения к модулям, которые просто находятся в другом каталоге файловой системы, недопустимо.

8.11. ВЫПОЛНЕНИЕ ПОДМОДУЛЯ ПАКЕТА В КАЧЕСТВЕ СЦЕНАРИЯ

Среда исполнения кода, разделенного на пакеты, отличается от среды исполнения простого сценария по нескольким аспектам. Среди них — имя содержащего пакета, подмодули и возможность относительного импорта (который работает только в пакете). Один из аспектов, который перестает работать, — возможность выполнения Python прямо с исходным файлом пакета. Допустим, вы работаете над файлом `graphics/graph2d/plot2d.py` и добавляете в конец тестовый код:

```
# graphics/graph2d/plot2d.py
from ..primitive import lines, text

class Plot2D:
    ...

if __name__ == '__main__':
    print('Testing Plot2D')
    p = Plot2D()
    ...
```

При попытке выполнить этот файл напрямую произойдет фатальный сбой с сообщением о командах относительного импорта:


```
bash $ python3 graphics/graph2d/plot2d.py
Traceback (most recent call last):
  File "graphics/graph2d/plot2d.py", line 1, in <module>
    from ..primitive import line, text
ValueError: attempted relative import beyond top-level package
bash $
```

Перейти в каталог пакета и выполнить файл оттуда тоже не получится:

```
bash $ cd graphics/graph2d/
bash $ python3 plot2d.py
Traceback (most recent call last):
  File "plot2d.py", line 1, in <module>
    from ..primitive import line, text
ValueError: attempted relative import beyond top-level package
bash $
```

Для выполнения подмодуля как основного сценария передайте интерпретатору ключ `-m`:

```
bash $ python3 -m graphics.graph2d.plot2d
Testing Plot2D
bash $
```

`-m` назначает модуль или пакет основной программой. Python выполнит модуль в соответствующей среде для обеспечения правильной работы импорта. Многие встроенные пакеты Python обладают «секретной» функциональностью, которая может использоваться с помощью ключа `-m`. Один из самых известных примеров — команда `python3 -m http.server` для запуска веб-сервера из текущего каталога.

Вы можете предоставить такую же функциональность в ваших пакетах. Если имя для команды `python -m имя` соответствует каталогу пакета, Python проверяет наличие файла `__main__.py` в этом каталоге и выполняет его как сценарий.

8.12. УПРАВЛЕНИЕ ПРОСТРАНСТВОМ ИМЕН ПАКЕТА

В первую очередь пакеты служат контейнером верхнего уровня для кода. Иногда пользователи импортируют имя верхнего уровня и ничего более. Пример:

```
import graphics
```

В этой команде не указан никакой конкретный подмодуль. Она не открывает дополнительного доступа к другим частям пакета. Например, такой код не работает:

```
import graphics
graphics.primitive.fill.floodfill(img,x,y,color) # Ошибка!
```

При импорте только пакета верхнего уровня будет импортирован лишь соответствующий файл `__init__.py`. В нашем примере это `graphics/__init__.py`.

Главное предназначение файла `__init__.py` — построение и/или управление содержимым пространства имен пакета верхнего уровня. Часто это подразумевает импорт отдельных функций, классов и других объектов из подмодулей более низких уровней. Например, если пакет `graphics` в этом примере состоит из сотен низкоуровневых функций, но большинство этих подробностей инкапсулировано в нескольких высокоуровневых классах, файл `__init__.py` может ограничиться экспортом только этих классов:

```
# graphics/__init__.py

from .graph2d.plot2d import Plot2D
from .graph3d.plot3d import Plot3D
```

С таким файлом `__init__.py` имена `Plot2D` и `Plot3D` появляются на верхнем уровне пакета. Тогда пользователь сможет применить эти имена, как если бы модуль `graphics` был одноуровневым:

```
from graphics import Plot2D

plt = Plot2D(100, 100)
plt.clear()
...
```

Часто это удобнее для пользователя, ведь ему не нужно знать, как реально организован ваш код. В каком-то смысле вы накладываете более высокий уровень абстракции на структуру своего кода. Многие модули стандартной библиотеки Python построены так. Например, популярный модуль `collections` — это пакет. Файл `collections/__init__.py` объединяет определения из нескольких разных мест и дает их пользователю в виде одного консолидированного пространства имен.

8.13. УПРАВЛЕНИЕ ЭКСПОРТОМ ПАКЕТОВ

Одна из возможных проблем связана с взаимодействием между файлом `__init__.py` и низкоуровневыми подмодулями. Пользователя пакета могут интересовать только объекты и функции в пространстве имен пакета верхнего

уровня, а разработчик предпочитает разбить код на подмодули, удобные в сопровождении.

Для лучшего управления организационной сложностью подмодули пакета часто объявляют явный список экспорта, определяя переменную `__all__`. Это список имен, которые должны быть вынесены уровнем выше в пространстве имен пакета. Пример:

```
# graphics/graph2d/plot2d.py
```

```
__all__ = ['Plot2D']
```

```
class Plot2D:
```

```
    ...
```

Далее соответствующий файл `__init__.py` импортирует свои подмодули с использованием синтаксиса `*`:

```
# graphics/graph2d/__init__.py
```

```
# Загружаются только имена, явно перечисленные в переменных __all__  
from .plot2d import *
```

```
# __all__ распространяется на следующий уровень (если нужно)
```

```
__all__ = plot2d.__all__
```

Процесс повышения продолжается вплоть до файла `__init__.py` верхнего уровня. Пример:

```
# graphics/__init__.py
```

```
from .graph2d import *
```

```
from .graph3d import *
```

```
# Консолидация экспорта
```

```
__all__ = [  
    *graph2d.__all__,  
    *graph3d.__all__  
]
```

Каждый компонент пакета здесь явно определяет свои экспортируемые имена в переменной `__all__`. Затем файлы `__init__.py` распространяют экспорт вверх. На практике это может быть сложно, но такой подход позволяет избежать проблемы с жесткой привязкой конкретных имен экспорта в файл `__init__.py`. Вместо этого, если подмодуль хочет что-то экспортировать, его имя указывается в одном месте — в переменной `__all__`. После чего

оно волшебным образом распространяется вверх до своего законного места в пространстве имен пакета.

Хотя импортирование `*` в пользовательском коде обычно нежелательно, оно довольно часто встречается в файлах `__init__.py` пакетов. Это работает в пакетах, потому что они обычно более управляемы и автономны. Они руководствуются содержимым переменных `__all__`, а не бесшабашным подходом «а теперь давайте импортируем все».

8.14. ДАННЫЕ ПАКЕТОВ

Иногда пакет включает файлы данных, которые должны загружаться в программе (в отличие от файлов с исходным кодом). Внутри пакета переменная `__file__` дает информацию о местонахождении конкретного исходного файла.

Но пакеты усложняют ситуацию. Они могут распространяться в архивных ZIP-файлах или загружаться из нетипичных сред. Сама переменная `__file__` может быть ненадежной или неопределенной. В результате загрузка файла данных часто не сводится к простой передаче имени файла встроенной функции `open()` и чтению данных.

Для чтения данных пакета используется функция `pkgutil.get_data(накет, ресурс)`. Допустим, ваш пакет выглядит так:

```
mycode/
  resources/
    data.json
  __init__.py
  spam.py
  yow.py
```

Загрузить файл `data.json` из файла `spam.py` можно так:

```
# mycode/spam.py

import pkgutil
import json

def func():
    rawdata = pkgutil.get_data(__package__,
                               'resources/data.json')
    textdata = rawdata.decode('utf-8')
    data = json.loads(textdata)
    print(data)
```

Функция `get_data()` пытается найти заданный ресурс и возвращает его содержимое как необработанную последовательность байтов. В переменной `__package__` здесь хранится строка с именем содержащего пакета. Все дальнейшее декодирование (например, преобразование байтов в текст) и интерпретация остаются за вами. В этом примере данные декодируются и преобразуются из разметки JSON в словарь Python.

Пакеты не подходят для хранения огромных файлов данных. Их ресурсы обычно лучше ограничивать данными конфигурации и другими разнокалиберными данными для работы вашего пакета.

8.15. ОБЪЕКТЫ МОДУЛЕЙ

Модули — это первоклассные объекты. В табл. 8.1 находятся атрибуты многих модулей.

Таблица 8.1. Атрибуты модулей

| АТТРИБУТ | ОПИСАНИЕ |
|------------------------------|--|
| <code>__name__</code> | Полное имя модуля |
| <code>__doc__</code> | Строка документации |
| <code>__dict__</code> | Словарь модуля |
| <code>__file__</code> | Файл, в котором модуль определяется |
| <code>__package__</code> | Имя содержащегося пакета (если есть) |
| <code>__path__</code> | Список подкаталогов для поиска подмодулей пакета |
| <code>__annotations__</code> | Аннотации типов уровня модуля |

Атрибут `__dict__` содержит словарь, представляющий пространство имен модуля. В нем размещается все, что определяется в модуле.

`__name__` часто используется в сценариях. Проверки вида `if __name__ == '__main__':` используются, чтобы проверить, выполняется ли файл как автономная программа.

`__package__` содержит имя пакета (если он есть). Если атрибут задан, `__path__` содержит список каталогов, где будет производиться поиск подмодулей пакета. Обычно он содержит только один элемент с каталогом, где находится пакет. Иногда большие фреймворки оперируют с `__path__` и включают дополнительные каталоги для поддержки плагинов и других расширенных возможностей.

Не все атрибуты доступны во всех модулях. Например, у встроенных модулей может не быть установленного атрибута `__file__`. Точно так же атрибуты, относящиеся к пакету, не устанавливаются для модулей верхнего уровня (не содержащихся в пакете).

`__doc__` содержит строку документации модуля (если она есть). Она находится в первой команде файла. Атрибут `__annotations__` содержит словарь аннотаций типов уровня модуля. Он выглядит примерно так:

```
# mymodule.py
...
Строка документации
...
# Аннотации типов (помещаются в __annotations__)
x: int
y: float
...
```

Как и другие аннотации типов, аннотации уровня модуля никак не влияют на поведение Python и не определяют переменных. Это лишь метаданные, к которым при желании могут обратиться другие средства.

8.16. РАЗВЕРТЫВАНИЕ ПАКЕТОВ PYTHON

Последний рубеж модулей и пакетов — проблема распространения вашего кода. Это большая область, где последние годы ведутся активные разработки. Я не буду даже пытаться документировать процесс, который заведомо устареет к моменту прочтения этой книги. Вместо этого советую обращаться к документации по адресу: <https://packaging.python.org/tutorials/packaging-projects>.

Для повседневной разработки важнее всего поддерживать изоляцию вашего кода в автономном проекте. Весь ваш код должен быть в соответствующем пакете. Постарайтесь присвоить пакету уникальное имя, чтобы он не конфликтовал с другими возможными зависимостями. За выбором имени обращайтесь к пакету Python по адресу: <https://pypi.org>.

При выборе структуры кода стремитесь к простоте. Как вы уже видели, система модулей и пакетов поддерживает немало базовых возможностей. Для них найдется и время, и место, но начинать лучше не с них.

В самом простом и минимальном способе распространения «чистого» кода Python используется модуль `setuptools` или встроенный модуль `distutils`. Допустим, ваш код — это часть проекта, который выглядит примерно так:

```
spam-project/  
  README.txt  
  Documentation.txt  
  spam/      # Пакет с кодом  
    __init__.py  
    foo.py  
    bar.py  
  runspam.py # Сценарий, который должен выполняться  
              # командой python runspam.py
```

Для дистрибутива создайте файл `setup.py` в каталоге верхнего уровня (`spam-project/` в этом примере). Включите в него следующий код:

```
# setup.py  
from setuptools import setup  
  
setup(name="spam",  
      version="0.0",  
      packages=['spam'],  
      scripts=['runspam.py'],  
    )
```

При вызове `setup()` аргумент `packages` содержит список всех каталогов пакетов, а аргумент `scripts` — список файлов сценариев. Любые аргументы можно опустить, если они не поддерживаются вашим программным обеспечением (например, при отсутствии сценариев). Вызов `setup()` поддерживает множество других параметров, предоставляющих метаданные о пакете. Полный список доступен по адресу: <https://docs.python.org/3/distutils/apiref.html>.

Создания файла `setup.py` достаточно для дистрибутива вашего программного продукта. Для построения дистрибутива с исходным кодом введите следующую команду:

```
bash $ python setup.py sdist  
...  
bash $
```

Команда создает архивный файл (например, `spam-1.0.tar.gz` или `spam-1.0.zip`) в каталоге `spam/dist`. Этот файл вы будете передавать пользователям. Для установки программного продукта пользователь может воспользоваться специальной командой, например `pip`:

```
shell $ python3 -m pip install spam-1.0.tar.gz
```

Команда загружает программу в локальной установке Python и делает ее общедоступной. Код обычно устанавливается в каталоге `site-packages`

библиотеки Python. Чтобы узнать точное местонахождение этого каталога, проверьте значение `sys.path`. Сценарии обычно устанавливаются в одном каталоге с интерпретатором Python.

Если первая строка скрипта начинается с `#!` и содержит текст `python`, программа установки переписшет строку, чтобы указать на локальную установку Python. Если ваши сценарии были жестко запрограммированы на использование конкретного каталога, например `/usr/local/bin/python`, они должны работать и в других системах, где Python установлен в другом месте.

Важно, что описанный вариант использования `setuptools` абсолютно минимален. Более серьезные проекты могут включать расширения C/C++, сложные структуры пакетов, примеры и многое другое. Рассмотрение всех инструментов и возможных способов развертывания такого кода выходит за рамки книги. За самой актуальной информацией обращайтесь к ресурсам на сайтах <https://python.org> и <https://pypi.org>.

8.17. НАЧИНАЙТЕ С ПАКЕТА

Работу над новой программой легко начать с простого файла Python. Например, вы можете написать сценарий `program.py`, который станет отправной точкой. Такой способ неплохо подходит для временных и коротких задач, но со временем ваш сценарий начнет расти и дополняться новой функциональностью. В какой-то момент вы захотите разбить его на несколько файлов — и здесь часто начинаются проблемы.

Из-за этого вам стоит взять в привычку начинать работу над всеми программами в форме пакета. Например, вместо файла с именем `program.py` создайте каталог пакета с именем `program`:

```
program/  
    __init__.py  
    __main__.py
```

Включите начальный код в файл `__main__.py` и запустите свою программу командой вида `python -m program`. Для добавления нового кода включите в пакет новые файлы и используйте импорт относительно пакета. Преимущество использования пакета в том, что весь ваш код остается изолированным. Файлам можно присвоить любые имена, не беспокоясь о конфликтах с другими пакетами, модулями стандартной библиотеки или кодом ваших коллег. Хотя настройка пакета поначалу потребует чуть больше работы, она избавит вас от многих проблем позднее.

8.18. НАПОСЛЕДОК: БУДЬТЕ ПРОЩЕ

Система модулей и пакетов открывает куда больше возможностей для хитроумного шаманства, чем показано здесь. Чтобы получить представление о возможностях, обращайтесь к учебнику *Modules and Packages: Live and Let Die!*: <https://dabeaz.com/modulepackage/index.html>.

Но с учетом всех обстоятельств вам лучше обойтись *без* экзотических трюков с модулями. Управление модулями, пакетами и распространение программного кода всегда добавляли проблем в сообществе Python. И многие возникали из-за того, что разработчики пытались сделать что-то неочевидное с системой модулей. Не делайте так. Будьте проще и найдите в себе силы отказать своим коллегам, когда они предложат изменить систему импорта для достижения совместимости с блокчейном.

ГЛАВА 9

Ввод/вывод

Ввод и вывод — важнейшая часть любых программ. В этой главе описаны основы ввода/вывода в Python, включая кодирование данных, параметры командной строки, переменные среды, файловый ввод/вывод и сериализацию данных. Особое внимание уделяется средствам программирования и абстракциям, обеспечивающим правильную работу с вводом/выводом. В конце главы приводится сводка часто используемых модулей стандартной библиотеки, относящихся к области ввода/вывода.

9.1. ПРЕДСТАВЛЕНИЕ ДАННЫХ

Главная проблема ввода/вывода — окружающий мир. Для взаимодействия с ним данные должны быть правильно представлены, чтобы ими можно было управлять. На нижнем уровне Python работает с двумя основными типами данных: *байтами*, низкоуровневыми неинтерпретированными данными любого типа, и *текстом*, отображающим символы «Юникода».

Для представления байтов используются два встроенных типа: `bytes` и `bytearray`. `bytes` — это неизменяемая строка целочисленных значений байтов. `bytearray` — изменяемый массив байтов, который работает как комбинация байтовой строки и списка. Благодаря своей изменяемости он подходит для пошагового построения групп байтов, как при сборе данных из фрагментов. Следующий пример показывает некоторые возможности `bytes` и `bytearray`:

```
# Определение литерала bytes (обратите внимание на префикс b)
a = b'hello'
```

```
# Создание bytes по списку целых чисел
b = bytes([0x68, 0x65, 0x6c, 0x6c, 0x6f])
```

```
# Создание и заполнение bytearray по частям
```

```
c = bytearray()
c.extend(b'world') # c = bytearray(b'world')
c.append(0x21) # c = bytearray(b'world!')

# Обращение к значениям байтов
print(a[0]) # --> выводит 104

for x in b: # Выводит 104 101 108 108 111
    print(x)
```

При обращении к отдельным элементам объектов `bytes` и `bytearray` вы получаете целочисленные значения байтов, а не односимвольные байтовые строки. Этим они отличаются от текстовых строк, а такая ошибка часто встречается при работе с байтовыми данными.

Текст представляется типом данных `str` и сохраняется в массиве кодовых пунктов «Юникода»:

```
d = 'hello'    # Текст (Юникод)
len(d)         # --> 5
print(d[0])    # выводит 'h'
```

Python поддерживает строгое разделение между байтами и текстом. Между двумя типами никогда не выполняются автоматические преобразования. Сравнения между ними дают результат `False`, а любая операция, где байты смешиваются с текстом, приводит к ошибке:

```
a = b'hello'   # байты
b = 'hello'    # текст
c = 'world'    # текст
print(a == b)  # -> False
d = a + c      # TypeError: конкатенация строки с байтами невозможна
e = b + c      # -> 'helloworld' (оба операнда - строки)
```

Выполняя ввод/вывод, убедитесь, что вы работаете с правильным представлением данных. Во время работы с текстом используйте текстовые строки, а с двоичными данными используйте байты.

9.2. КОДИРОВАНИЕ И ДЕКОДИРОВАНИЕ ТЕКСТА

Если вы работаете с текстом, все прочитанные из ввода данные должны быть декодированы, а все данные, записанные в вывод, — закодированы. Явное преобразование между текстами и байтами выполняется методами

`encode(текст [,ошибки])` и `decode(байты [,ошибки])` текстовых и байтовых объектов соответственно:

```
a = 'hello' # Текст
b = a.encode('utf-8') # Кодирование в байты
c = b'world' # Байты
d = c.decode('utf-8') # Декодирование в текст
```

`encode()` и `decode()` должно передаваться название кодировки, такое как `'utf-8'` или `'latin-1'`. В табл. 9.1 перечислены самые популярные кодировки.

Таблица 9.1. Популярные кодировки

| НАЗВАНИЕ КОДИРОВКИ | ОПИСАНИЕ |
|--------------------|--|
| 'ascii' | Значения символов в диапазоне [0x00, 0x7f] |
| 'latin1' | Значения символов в диапазоне [0x00, 0xff]. Также известна как <code>'iso-8859-1'</code> |
| 'utf-8' | Кодировка переменной длины, обеспечивающая представление всех символов «Юникода» |
| 'cp1252' | Стандартная кодировка текста для Windows |
| 'macroman' | Стандартная кодировка текста для Macintosh |

Методы кодирования могут получать необязательный аргумент, задающий поведение при обнаружении ошибок. Аргумент содержит одно из значений, перечисленных в табл. 9.2.

Таблица 9.2. Политики обработки ошибок

| ЗНАЧЕНИЕ | ОПИСАНИЕ |
|---------------------|---|
| 'strict' | Выдает исключение <code>UnicodeError</code> при ошибках кодирования и декодирования (используется по умолчанию) |
| 'ignore' | Игнорирует недопустимые символы |
| 'replace' | Заменяет недопустимые символы символом-заменителем (U+FFFD в «Юникоде», <code>b'?'</code> в байтовых данных) |
| 'backslashreplace' | Заменяет каждый недопустимый символ служебной последовательностью символа в Python. Например, символ U+1234 заменяется <code>'\u1234'</code> (только кодирование) |
| 'xmlcharrefreplace' | Заменяет каждый недопустимый символ символьной ссылкой XML. Например, символ U+1234 заменяется <code>'&#4660;'</code> (только кодирование) |
| 'surrogateescape' | Заменяет каждый недопустимый байт <code>'\xhh'</code> на U+DChh при декодировании, заменяет U+DChh байтом <code>'\xhh'</code> при кодировании |

Политики 'backslashreplace' и 'xmlcharrefreplace' — непредставимые символы в форме, позволяющей просматривать их как простой ASCII-текст или символьные ссылки XML. Это может пригодиться при отладке.

Политика обработки ошибок 'surrogateescape' позволяет вырожденным байтовым данным (не следующим ожидаемым правилам кодирования) пережить цикл кодирования/декодирования независимо от используемой кодировки текста. Говоря конкретнее, `s.decode(enc, 'surrogateescape').encode(enc, 'surrogateescape') == s`. Сохранение данных полезно в некоторых разновидностях системных интерфейсов, где кодирование текста ожидается, но не может гарантироваться из-за проблем, неподконтрольных Python. Вместо уничтожения данных с некорректной кодировкой, Python встраивает их как есть в суррогатной кодировке. Пример такого поведения с некорректно закодированной строкой UTF-8:

```
>>> a = b'Spicy Jalape\x10' # Invalid UTF-8
>>> a.decode('utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1
in position 12: invalid continuation byte
>>> a.decode('utf-8', 'surrogateescape')
'Spicy Jalape\uudcf10'
>>> # Encode the resulting string back into bytes
>>> _.encode('utf-8', 'surrogateescape')
b'Spicy Jalape\x10'
>>>
```

9.3. ФОРМАТИРОВАНИЕ ТЕКСТА И БАЙТОВ

Частым источником проблем при работе с текстовыми и байтовыми строками становятся преобразования и форматирование строк. Например, преобразование числа с плавающей точкой в строку с заданной шириной и точностью. Для форматирования одиночных значений используется функция `format()`:

```
x = 123.456
format(x, '0.2f')    # '123.46'
format(x, '10.4f')   # ' 123.4560'
format(x, '<*10.2f')  # '123.46****'
```

Второй аргумент `format()` содержит спецификатор формата. Общий формат спецификатора имеет вид `[[заполнение][выравнивание]][знак][0][ширина][,][.точность][min]`. Каждая часть в квадратных скобках `[]` необязательна. *Ширина* задает минимальную ширину поля, а *выравнивание* содержит

один из знаков <, > или ^ для выравнивания по левому, правому краю или центру поля соответственно.

Необязательный символ *заполнение* используется для заполнения свободных позиций до заданной ширины:

```
name = 'Elwood'
r = format(name, '<10') # r = 'Elwood '
r = format(name, '>10') # r = ' Elwood'
r = format(name, '^10') # r = ' Elwood '
r = format(name, '**10') # r = '**Elwood**'
```

Спецификатор *тип* задает тип данных. В табл. 9.3 перечислены поддерживаемые коды форматирования. Если код не задан, то по умолчанию используется s для строк, d для целых чисел и f для чисел с плавающей точкой.

Таблица 9.3. Коды форматирования

| СИМВОЛ | ВЫХОДНОЙ ФОРМАТ |
|--------|--|
| d | Десятичное или длинное целое |
| b | Двоичное или длинное целое |
| o | Восьмеричное или длинное целое |
| x | Шестнадцатеричное или длинное целое |
| X | Шестнадцатеричное целое (буквы верхнего регистра) |
| f, F | Число с плавающей точкой в формате [-]m.dddddd |
| e | Число с плавающей точкой в формате [-]m.dddddd \pm xx |
| E | Число с плавающей точкой в формате [-]m.ddddddE \pm xx |
| g | Для заданной точности $p \geq 1$ округляет число до p значащих цифр и затем форматирует результат либо в формате с фиксированной точкой, либо в научном представлении, в зависимости от его величины |
| G | То же, что и g, за исключением переключения на E, если число становится слишком большим |
| n | То же, что g, за исключением того, что символ разделителя дробной части определяется текущими настройками локального контекста |
| % | Число умножается на 100 и выводится в формате f, после чего выводится знак % |
| s | Строка или любой объект. Код форматирования использует str() для генерации строк |
| c | Одиночный символ |

Часть *знак* спецификатора формата содержит одно из следующих значений: +, - или пробел; + означает, что все числа должны выводиться с начальным знаком; - используется по умолчанию и добавляет знак только к отрицательным числам. Пробел добавляет начальный пробел к положительным числам.

Между шириной и точностью может находиться необязательная запятая (,). Она добавляет символ — разделитель групп битов:

```
x = 123456.78
format(x, '16,.2f')    # ' 123,456.78'
```

Точность в спецификаторе задает число знаков точности для дробных чисел. Если в поле *ширина* для числа добавлен начальный 0, то числовые значения дополняются начальными нулями для заполнения выделенного места. Несколько примеров форматирования чисел разных видов:

```
x = 42
r = format(x, '10d')    # r = ' 42'
r = format(x, '10x')    # r = ' 2a'
r = format(x, '10b')    # r = ' 101010'
r = format(x, '010b')   # r = '0000101010'
y = 3.1415926
r = format(y, '10.2f')   # r = ' 3.14'
r = format(y, '10.2e')   # r = ' 3.14e+00'
r = format(y, '+10.2f')  # r = ' +3.14'
r = format(y, '+010.2f') # r = '+000003.14'
r = format(y, '+10.2%')  # r = ' +314.16%'
```

Для более сложного форматирования строк используются f-строки:

```
x = 123.456
f'Value is {x:0.2f}'    # 'Value is 123.46'
f'Value is {x:10.4f}'   # 'Value is 123.4560'
f'Value is {2*x:<10.2f}' # 'Value is 246.91****'
```

В f-строке текст вида {*выражение:спецификация*} заменяется значением `format(выражение, спецификация)`. *Выражение* может быть произвольным, без символов {, } или \. Части спецификатора могут задаваться другими выражениями:

```
y = 3.1415926
width = 8
precision=3

r = f'{y:{width}.{precision}f}' # r = ' 3.142'
```

Если *выражение* заканчивается на `=`, то его литеральный текст тоже включается в результат:

```
x = 123.456

f'{x=:0.2f}' # 'x=123.46'
f'{2*x=:0.2f}' # '2*x=246.91'
```

Если присоединить к значению `!r`, форматирование применяется к выводу `repr()`. При использовании `!s` форматирование применяется к выводу `str()`:

```
f'{x!r:spec}' # Вызывает (repr(x).__format__('spec'))
f'{x!s:spec}' # Вызывает (str(x).__format__('spec'))
```

Вместо f-строк можно воспользоваться методом `.format()` строк:

```
x = 123.456
'Value is {:0.2f}'.format(x)          # 'Value is 123.46'
'Value is {0:10.2f}'.format(x)        # 'Value is 123.4560'
'Value is {val:<*10.2f}'.format(val=x) # 'Value is 123.46****'
```

Со строкой, отформатированной вызовом `.format()`, текст в форме *{аргумент:спецификация}* заменяется значением `format(аргумент, спецификация)`. Здесь *аргумент* относится к одному из аргументов, переданных методу `format()`. В случае его опущения аргументы обрабатываются по порядку:

```
name = 'IBM'
shares = 50
price = 490.1
r = '{:>10s} {:10d} {:10.2f}'.format(name, shares, price)
# r = '          IBM           50      490.10'
```

Аргумент может обозначать и конкретный номер или имя аргумента:

```
tag = 'p'
text = 'hello world'

r = '<{0}>{1}</{0}>'.format(tag, text) # r = '<p>hello world</p>'
r = '<{tag}>{text}</{tag}>'.format(tag='p', text='hello world')
```

В отличие от f-строк, часть *аргумент* в спецификаторе не может быть произвольным выражением. Поэтому ее выразительные возможности небольшие. Метод `format()` может выполнять ограниченный поиск атрибутов, индексирование и вложенные подстановки:


```

y = 3.1415926
width = 8
precision=3

r = 'Value is {0:{1}.{2}f}'.format(y, width, precision)

d = {
    'name': 'IBM',
    'shares': 50,
    'price': 490.1
}
r = '{0[shares]:d} shares of {0[name]} at {0[price]:0.2f}'.format(d)
# r = '50 shares of IBM at 490.10'

```

Экземпляры `bytes` и `bytearray` могут форматироваться оператором `%`. Его семантика строится по образцу функции `sprintf()` из языка C. Несколько примеров:

```

name = b'ACME'
x = 123.456

b'Value is %0.2f' % x          # b'The value is 123.46'
bytearray(b'Value is %0.2f') % x # b'Value is 123.46'
b'%s = %0.2f' % (name, x)     # b'ACME = 123.46'

```

С таким форматированием последовательности вида `%c` заменяются по порядку значениями из кортежа, передаваемого во втором операнде оператора `%`. Базовые коды форматирования (`d`, `f`, `s` и т. д.) остаются теми же, что использовались функцией `format()`. Но более сложных возможностей либо нет, либо они слегка изменяются. Например, для настройки выравнивания используется символ `-`:

```

x = 123.456
b'%10.2f' % x   # b' 123.46'
b'%-10.2f' % x  # b'123.46 '

```

Код форматирования `%r` генерирует вывод `ascii()`. Он может пригодиться для отладки и ведения журнала.

Работая с байтами, помните: текстовые строки не поддерживаются — они должны кодироваться явно.

```

name = 'Dave'

b'Hello %s' % name          # TypeError!
b'Hello %s' % name.encode('utf-8') # Ok

```

Эта разновидность форматирования может использоваться и с текстовыми строками. Но она считается признаком старого стиля программирования. Тем не менее она еще встречается в некоторых библиотеках. Например, так формируются сообщения, производимые модулем `logging`:

```
import logging
log = logging.getLogger(__name__)

log.debug('%s got %d', name, value) # '%s got %d' % (name, value)
```

`logging` кратко описан позднее, в подразделе 9.15.12.

9.4. ЧТЕНИЕ ПАРАМЕТРОВ КОМАНДНОЙ СТРОКИ

При запуске Python параметры командной строки помещаются в список `sys.argv` в виде текстовых строк. Первый элемент содержит имя программы, а в дальнейших содержатся параметры, переданные в командной строке после имени программы. Следующая программа — минимальный прототип ручной обработки аргументов командной строки:

```
def main(argv):
    if len(argv) != 3:
        raise SystemExit(
            f'Usage : python {argv[0]} inputfile outputfile\n')
    inputfile = argv[1]
    outputfile = argv[2]
    ...

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Для улучшения эффективности организации кода, тестирования и по другим причинам стоит написать специализированную функцию `main()`. Она получает параметры командной строки (если они есть) в виде списка — вместо прямого чтения `sys.argv`. Включите небольшой фрагмент кода в конец своей программы, чтобы передать параметры командной строки в функцию `main()`.

`sys.argv[0]` содержит имя выполняемого сценария. Вывод содержательного справочного сообщения и выдача `SystemExit` — стандартная практика для сценариев командной строки, которые хотят сообщить об ошибке.

Хотя в простых сценариях параметры командной строки можно обрабатывать вручную, при более сложной обработке можно воспользоваться модулем `argparse`. Пример:

```
import argparse

def main(argv):
    p = argparse.ArgumentParser(description="This is some program")

    # Позиционный аргумент
    p.add_argument("infile")

    # Параметр для передачи аргумента
    p.add_argument("-o", "--output", action="store")

    # Параметр, устанавливающий логический флаг
    p.add_argument("-d", "--debug", action="store_true", default=False)

    # Разбор командной строки
    args = p.parse_args(args=argv)

    # Получение настроек
    infile = args.infile
    output = args.output
    debugmode = args.debug

    print(infile, output, debugmode)

if __name__ == '__main__':
    import sys
    main(sys.argv[1:])
```

Этот пример показывает простейшее использование модуля `argparse`. В документации стандартной библиотеки более сложные примеры. Есть и сторонние модули (например `click` и `docopt`), которые могут упростить написание более сложных парсеров командной строки.

Наконец, параметры командной строки могут передаваться Python в неподходящей кодировке. Такие аргументы будут приняты, но закодированы в режиме `'surrogateescape'` (см. раздел 9.2). Нужно учитывать это, если такие аргументы позднее включаются в какой-либо текстовый вывод и для вас крайне важно предотвратить сбой. С другой стороны, это может быть не критично — не усложняйте свой код обработкой неважных пограничных случаев.

9.5. ПЕРЕМЕННЫЕ СРЕДЫ

Иногда данные передаются программе через переменные среды в командной строке. Например, программа Python может запускаться командой вроде `env`:

```
bash $ env SOMEVAR=somevalue python3 somescript.py
```

К переменным среды можно обращаться как к текстовым строкам в отображении `os.environ`:

```
import os
path = os.environ['PATH']
user = os.environ['USER']
editor = os.environ['EDITOR']
val = os.environ['SOMEVAR']
... и т. д. ...
```

Чтобы изменить переменные среды, задайте значение переменной `os.environ`. Пример:

```
os.environ['NAME'] = 'VALUE'
```

Изменения в `os.environ` распространяются как на работающую программу, так и на все подпроцессы, созданные позднее, например созданные модулем `subprocess`.

Как и в случае с параметрами командной строки, неправильно закодированные переменные среды могут привести к появлению строк, использующих политику обработки ошибок `'surrogateescape'`.

9.6. ФАЙЛЫ И ОБЪЕКТЫ ФАЙЛОВ

Для открытия файла используйте встроенную функцию `open()`. Обычно `open()` передается имя файла и режим открытия. Она часто используется и вместе с командой `with` как менеджер контекста. Несколько примеров стандартных схем работы с файлами:

```
# Чтение всего содержимого файла в строку
with open('filename.txt', 'rt') as file:
    data = file.read()

# Чтение файла по строкам
with open('filename.txt', 'rt') as file:
    for line in file:
        ...
```

```
# Запись в текстовый файл
with open('out.txt', 'wt') as file:
    file.write('Some output\n')
    print('More output', file=file)
```

В основном использование `open()` достаточно прямолинейно. Вы передаете имя открываемого файла и режим открытия:

```
open('name.txt')          # Открывает "name.txt" для чтения
open('name.txt', 'rt')    # Открывает "name.txt" для чтения (то же)
open('name.txt', 'wt')    # Открывает "name.txt" для записи
open('data.bin', 'rb')    # Чтение в двоичном режиме
open('data.bin', 'wb')    # Запись в двоичном режиме
```

В большинстве программ для работы с файлами хватит этих простых примеров. Но есть ряд специальных случаев и более экзотических возможностей `open()`, о которых желательно знать. В нескольких ближайших разделах функция `open()` и файловый ввод/вывод рассматриваются подробнее.

9.6.1. Имена файлов

Чтобы открыть файл, нужно передать `open()` его имя. Это может быть как абсолютное имя вида `'/Users/guido/Desktop/files/old/data.csv'`, так и относительное вида `'data.csv'` или `'..\old\data.csv'`. Для относительных имен местонахождение файла определяется относительно текущего рабочего каталога, возвращаемого вызовом `os.getcwd()`. Его можно сменить вызовом `os.chdir(newdir)`.

Само имя тоже может быть закодировано в разных формах. Если это текстовая строка, то имя интерпретируется в соответствии с текстовой кодировкой, возвращаемой вызовом `sys.getfilesystemencoding()`, до передачи управляющей операционной системе. Если имя файла — байтовая строка, она остается незакодированной и передается как есть. Второй вариант может быть полезен, если вы пишете программы, которые должны обрабатывать возможность вырожденных или неправильно закодированных имен файлов. Вместо передачи имени файла в виде текста можно передать его низкоуровневое двоичное представление. Это может показаться неясным пограничным случаем, но Python обычно используется для написания скриптов системного уровня, управляющих файловой системой. Злоупотребления файловой системой — стандартный прием, используемый хакерами, для того чтобы замести следы или нарушить работу системы.

Именем может быть любой объект, реализующий специальный метод `__fspath__()`. `__fspath__()` должен вернуть текст или объект `bytes`,

соответствующий реальному имени. Этот механизм обеспечивает работу стандартных библиотечных модулей, например `pathlib`:

```
>>> from pathlib import Path
>>> p = Path('Data/portfolio.csv')
>>> p.__fspath__()
'Data/portfolio.csv'
>>>
```

Теоретически вы можете создать свой объект `Path`, который работает с `open()`. Для этого он должен реализовать `__fspath__()`, так чтобы в результате он давал правильное имя файла в системе.

Наконец, имена файлов могут передаваться в виде низкоуровневых целочисленных дескрипторов файлов. Для этого файл уже должен быть открыт в системе. Например, он может соответствовать сетевому сокету, каналу или другому системному ресурсу, предоставляющему дескриптор файла. В следующем примере файл открывается напрямую средствами модуля `os` и преобразуется в объект файла:

```
>>> import os
>>> fd = os.open('/etc/passwd', os.O_RDONLY) # Целочисленный дескриптор
                                           # файла
>>> fd
3
>>> file = open(fd, 'rt') # Объект файла
>>> file
<_io.TextIOWrapper name=3 mode='rt' encoding='UTF-8'>
>>> data = file.read()
>>>
```

При таком открытии существующего дескриптора файла метод `close()` возвращаемого файла закроет используемый дескриптор. Во избежание этого передайте `open()` аргумент `closefd=False`:

```
file = open(fd, 'rt', closefd=False)
```

9.6.2. Режимы открытия файлов

При открытии файла нужно указать режим: `'r'` для чтения, `'w'` для записи и `'a'` для присоединения. В режиме `'w'` любой существующий файл заменяется новым содержимым. Режим `'a'` открывает файл для записи и устанавливает файловый указатель в конце, чтобы к нему можно было присоединить новые данные.

Специальный режим файла 'x' может использоваться для записи в файл, но только если он еще не существует. Это полезный механизм предотвращения случайной перезаписи существующих данных. Если в этом режиме файл уже есть, выдается исключение `FileExistsError`.

В Python есть четкие различия между текстовыми и двоичными данными. Чтобы задать вид данных, присоедините символ 't' (текст) или 'b' (двоичные данные) к режиму открытия файла. Например, в 'rt' файл открывается для чтения в текстовом режиме, а в 'rb' — для чтения в двоичном режиме. Режим определяет, какие данные будут возвращаться такими методами, как `f.read()`. В текстовом режиме возвращаются строки, а в двоичном — байты.

Двоичные файлы можно открыть для обновления «на месте». Для этого добавляется символ «плюс» (+): например, 'rb+' или 'wb+'. Когда файл открывается для обновления, вы можете выполнять как операции ввода, так и операции вывода, если все операции вывода записывают свои данные до любых последующих операций ввода. Если файл открывается в режиме 'wb+', его длина сначала обнуляется. Режим обновления часто используется для предоставления произвольного доступа к чтению/записи содержимого файла в сочетании с операциями позиционирования.

9.6.3. Буферизация ввода/вывода

По умолчанию файлы открываются с включенной буферизацией ввода/вывода. С буферизацией операции ввода/вывода выполняются большими блоками для предотвращения лишних системных вызовов. Например, операции записи начинают заполнять внутренний буфер в памяти, а вывод выполняется только при заполнении буфера. Чтобы изменить это поведение, передайте аргумент `buffering` функции `open()`:

```
# Файл открывается в двоичном режиме без буферизации ввода/вывода
with open('data.bin', 'wb', buffering=0) as file:
    file.write(data)
    ...
```

Значение 0 включает небуферизованный ввод/вывод. Оно допустимо только для файлов двоичного режима. Значение 1 включает построчную буферизацию и обычно важно только для файлов текстового режима. Любое другое положительное значение задает размер буфера (в байтах).

Если значение `buffering` не указано, поведение по умолчанию зависит от типа файла. Если это обычный файл на диске, буферизация выполняется по блокам, а размер буфера задается равным `io.DEFAULT_BUFFER_SIZE`. Обычно

это значение кратно 4096 байтам и может изменяться в зависимости от системы. Если это интерактивный терминал, используется построчная буферизация.

Для обычных программ буферизация ввода/вывода обычно не создает серьезных проблем. Но она может влиять на приложения с активными взаимодействиями между процессами. Например, с двумя взаимодействующими подпроцессами иногда возникает взаимная блокировка из-за проблем внутренней буферизации. Допустим, один процесс записывает данные в буфер, но получатель не видит их, потому что буфер остается незаписанным. Такие проблемы могут решаться либо небуферизованным вводом/выводом, либо явным вызовом `flush()` для соответствующего файла:

```
file.write(data)
file.write(data)
...
file.flush() # Обеспечивает запись всех данных из буферов
```

9.6.4. Кодировка текстового режима

Для файлов, открываемых в текстовом режиме, можно дополнительно задать кодировку и политику обработки ошибок. С этой целью используются аргументы `encoding` и `errors`:

```
with open('file.txt', 'rt',
          encoding='utf-8', errors='replace') as file:
    data = file.read()
```

Значения, передаваемые в `encoding` и `errors`, имеют такой же смысл, как для методов `encode()` и `decode()` для строк и байтов соответственно.

Кодировка текста по умолчанию определяется значением `sys.getdefaultencoding()` и может изменяться в зависимости от системы. Если кодировка известна заранее, часто лучше явно указать ее, даже если она совпадает с кодировкой по умолчанию в вашей системе.

9.6.5. Обработка строк текста в текстовом режиме

Одна из трудностей при работе с текстовыми файлами связана с кодированием символов новой строки. Они кодируются в виде `'\n'`, `'\r\n'` или `'\r'` в зависимости от операционной системы. Например, `'\n'` в UNIX и `'\r\n'` в Windows. По умолчанию Python при чтении преобразует все эти

завершители строк в стандартный символ `'\n'`. При записи символы новой строки преобразуются обратно в завершители строк, используемые в системе. В документации Python этот обратный вызов иногда называется режимом универсальных новых строк.

Поведение новой строки можно изменить, передавая аргумент `newline` при вызове `open()`:

```
# Потребовать именно '\r\n' и оставить без изменений
file = open('somefile.txt', 'rt', newline='\r\n')
```

Значение `newline=None` включает поведение обработки строк по умолчанию, при котором все завершители строк преобразуются в стандартный символ `'\n'`. Присваивание `newline=''` заставляет Python распознавать все завершители строк, но блокирует этап преобразования. Если строки завершаются `'\r\n'`, то комбинация `'\r\n'` останется во входных данных в неизменном виде. При передаче значения `'\n'`, `'\r'` или `'\r\n'` она становится предполагаемым завершителем строк.

9.7. УРОВНИ АБСТРАКЦИИ ВВОДА/ВЫВОДА

Функция `open()` служит высокоуровневой фабричной функцией для создания экземпляров разных классов ввода/вывода. Эти классы воплощают разные режимы открытия файлов, кодировки и варианты поведения буферизации. Они имеют многоуровневую структуру. В модуле `io` определяются следующие классы.

```
FileIO(filename, mode='r', closefd=True, opener=None)
```

Открывает файл для низкоуровневого небуферизованного двоичного ввода/вывода. В аргументе `filename` передается любое действительное имя файла, которое принимается функцией `open()`. Другие аргументы имеют такой же смысл, как и для `open()`.

```
BufferedReader(file [, buffer_size])
BufferedWriter(file [, buffer_size])
BufferedRandom(file [, buffer_size])
```

Реализует буферизованный двоичный уровень ввода/вывода для файла. Аргумент `file` — экземпляр `FileIO`. `buffer_size` задает внутренний размер буфера. Выбор класса зависит от вида операций с файлом — чтение, запись или обновление данных. Необязательный аргумент `buffer_size` задает размер внутреннего буфера.

```
TextIOWrapper(buffered, [encoding, [errors [, newline [, line_buffering [,
write_through]]]])
```

Реализует ввод/вывод в текстовом режиме. `buffered` содержит буферизованный файл двоичного режима, например `BufferedReader` или `BufferedWriter`. `encoding`, `errors` и `newline` имеют такой же смысл, как для `open()`. `line_buffering` содержит логический флаг, который включает запись буфера ввода/вывода по символам новой строки (`False` по умолчанию). `write_through` содержит логический флаг, включающий запись при всех операциях (`False` по умолчанию).

Следующий пример показывает поуровневое конструирование файла текстового режима:

```
>>> raw = io.FileIO('filename.txt', 'r') # Низкоуровневый двоичный режим
>>> buffer = io.BufferedReader(raw)      # Двоичное буферизованное чтение
>>> file = io.TextIOWrapper(buffer, encoding='utf-8') # Текстовый режим
>>>
```

Обычно вам не нужно конструировать уровни вручную: встроенная функция `open()` делает всю работу. Но если у вас уже есть существующий объект файла, и вы хотите как-то изменить его обработку, вы можете работать с уровнями, как показано выше.

Для исключения используется метод `detach()` файла.

Вот как можно преобразовать существующий файл текстового режима в файл двоичного режима:

```
f = open('something.txt', 'rt') # Файл текстового режима
fb = f.detach() # Отсоединение нижележащего файла двоичного режима
data = fb.read() # Возвращает байты
```

9.7.1. Методы файлов

Точный тип объекта, возвращаемого `open()`, зависит от комбинации режима файла и переданных параметров буферизации. Полученный объект файла поддерживает методы из табл. 9.4.

Методы `readable()`, `writable()` и `seekable()` проверяют поддерживаемую функциональность и режимы файлов. `read()` возвращает весь файл в виде строки, если необязательный параметр не задает максимальное число символов. `readline()` возвращает следующую строку ввода, включая завершающий символ новой строки. `readlines()` возвращает весь входной файл в виде списка строк. `readline()` может получать необязательную максимальную

длину строки `n`. Если длина прочитанной строки превышает `n` символов, возвращаются первые `n` символов.

Таблица 9.4. Методы файлов

| МЕТОД | ОПИСАНИЕ |
|---------------------------------------|--|
| <code>f.readable()</code> | Возвращает <code>True</code> , если файл поддерживает чтение |
| <code>f.read([n])</code> | Возвращает не более <code>n</code> байт |
| <code>f.readline([n])</code> | Читает одну строку ввода длиной до <code>n</code> символов. Если значение <code>n</code> не указано, метод читает строку |
| <code>f.readlines([size])</code> | Читает все строки и возвращает список. Необязательный аргумент <code>size</code> задает приблизительное количество символов, которые будут прочитаны из файла до остановки |
| <code>f.readinto(buffer)</code> | Читает данные в буфер, находящийся в памяти |
| <code>f.writable()</code> | Возвращает <code>True</code> , если в файл возможна запись |
| <code>f.write(s)</code> | Записывает строку <code>s</code> |
| <code>f.writelines(lines)</code> | Записывает все строки в итерируемый объект <code>lines</code> |
| <code>f.close()</code> | Закрывает файл |
| <code>f.seekable()</code> | Возвращает <code>True</code> , если файл поддерживает позиционирование с произвольным доступом |
| <code>f.tell()</code> | Возвращает текущий файловый указатель |
| <code>f.seek(offset [, where])</code> | Позиционирует файловый указатель в новую позицию |
| <code>f.isatty()</code> | Возвращает <code>True</code> , если <code>f</code> служит интерактивным терминалом |
| <code>f.flush()</code> | Записывает на диск выходные буферы |
| <code>f.truncate([size])</code> | Усекает файл до размера не более <code>size</code> |
| <code>f.fileno()</code> | Возвращает целочисленный дескриптор файла |

Оставшиеся данные строки не теряются и будут возвращены в последующих операциях чтения. Метод `readlines()` получает параметр `size`, задающий приблизительное количество символов, которое будет прочитано перед остановкой. Их фактическое количество может быть больше в зависимости от объема уже буферизованных данных. `readinto()` используется для предотвращения копирования памяти, он будет рассмотрен ниже.

`read()` и `readline()` обозначают конец файла (EOF) возвращением пустой строки. Следующий код показывает выявление условия EOF:

```
while True:
    line = file.readline()
    if not line: # EOF
        break
    команды
    ...
```

Этот код можно записать так:

```
while (line:=file.readline()):
    команды
    ...
```

Удобный способ чтения всех строк из файла основан на использовании перебора в цикле `for`:

```
for line in file: # Перебрать все строки в файле
    # Обработать строку
    ...
```

Метод `write()` записывает данные в файл, а `writelines()` — в итерируемый объект со строками. `write()` и `writelines()` не добавляют в вывод символы новой строки. Весь сгенерированный вами вывод должен содержать все необходимое форматирование.

Во внутреннем представлении каждый объект открытого файла содержит файловый указатель — смещение в байтах, с которым будет выполняться следующая операция чтения или записи. Метод `tell()` возвращает текущее значение файлового указателя. `seek(offset [,whence])` используется для случайного обращения к части файла по целочисленному смещению и правилу размещения `whence`. Если `whence` содержит `os.SEEK_SET` (по умолчанию), `seek()` предполагает, что смещение задается относительно начала файла. Если `whence` содержит `os.SEEK_CUR`, позиция смещается относительно текущей. Если `whence` содержит `os.SEEK_END`, смещение задается относительно конца файла.

`fileno()` возвращает целочисленный дескриптор файла и иногда используется в низкоуровневых операциях ввода/вывода в некоторых библиотечных модулях. Например, `fcntl` использует дескриптор файла для предоставления низкоуровневых операций управления файлами в системах UNIX.

`readinto()` используется для чтения данных в непрерывные буферы в памяти. Он обычно используется в сочетании с такими специализированными библиотеками, как `numpy`. Например, для чтения данных прямо в память, выделенную для числового массива.

Объекты файлов содержат атрибуты, доступные только для чтения. Они перечислены в табл. 9.5.

Таблица 9.5. Атрибуты файлов

| АТРИБУТ | ОПИСАНИЕ |
|------------------------------|---|
| <code>f.closed</code> | Логический признак состояния файла: <code>False</code> , если файл открыт, или <code>True</code> для закрытого файла |
| <code>f.mode</code> | Режим ввода/вывода для файла |
| <code>f.name</code> | Имя файла, если он был создан вызовом <code>open()</code> . Иначе это будет строка, обозначающая источник файла |
| <code>f.newlines</code> | Представление новой строки, используемое в файле. Содержит <code>None</code> , если нет символов новой строки, строк <code>'\n'</code> , <code>'\r'</code> или <code>'\r\n'</code> , или кортеж со всеми обнаруженными кодировками новой строки |
| <code>f.encoding</code> | Строка с кодировкой файла, если есть (например, <code>'latin-1'</code> или <code>'utf-8'</code>). Содержит <code>None</code> , если кодировка не используется |
| <code>f.errors</code> | Политика обработки ошибок |
| <code>f.write_through</code> | Логическое значение, указывающее, передаются ли при записи данные в нижележащий двоичный файл без буферизации |

9.8. СТАНДАРТНЫЙ ВВОД, ВЫВОД И ПОТОК ОШИБОК

Интерпретатор предоставляет три стандартных похожих на файлы объекта — стандартный ввод, вывод и поток ошибок, доступные по именам `sys.stdin`, `sys.stdout` и `sys.stderr` соответственно. `stdin` — объект файла, представляющий поток входных символов, передаваемых интерпретатору. `stdout` — объект файла, получающий вывод, производимый `print()`, а `stderr` — файл, получающий сообщения об ошибках. Обычно `stdin` соответствует клавиатуре пользователя, а `stdout` и `stderr` связаны с текстом на экране.

Методы из прошлого раздела могут применяться для выполнения ввода/вывода пользователем. Следующий код осуществляет запись в стандартный вывод и читает строку данных из стандартного ввода:

```
import sys
sys.stdout.write("Enter your name: ")
name = sys.stdin.readline()
```

Встроенная функция `input(prompt)` может прочитать строку текста из `stdin` и вывести необязательное приглашение:

```
name = input("Enter your name: ")
```

Строки, прочитанные функцией `input()`, не включают завершающий символ новой строки. Этим функция отличается от прямого чтения из `sys.stdin`, когда символы новой строки добавляются во входной текст.

Значения `sys.stdout`, `sys.stdin` и `sys.stderr` можно заменять другими объектами файлов. Тогда функции `print()` и `input()` используют новые значения. Если исходное значение `sys.stdout` понадобится восстановить, его следует сначала сохранить. Исходные значения `sys.stdout`, `sys.stdin` и `sys.stderr` на момент запуска интерпретатора тоже доступны в `sys.__stdout__`, `sys.__stdin__` и `sys.__stderr__`.

9.9. КАТАЛОГИ

Для получения списка содержимого каталога используйте функцию `os.listdir(pathname)`. Вывод списка имен файлов в каталоге может быть выполнен так:

```
import os

names = os.listdir('dirname')
for name in names:
    print(name)
```

Имена, возвращаемые `listdir()`, обычно декодируются в соответствии с кодировкой, возвращаемой `sys.getfilesystemencoding()`. Если исходный путь задается в виде байтов, имена файлов возвращаются как не декодированные строки байтов:

```
import os

# Вернуть низкоуровневые не декодированные имена
names = os.listdir(b'dirname')
```

Полезная операция, связанная с обработкой содержимого каталогов, — отбор файлов по шаблону. Для этого подойдут модули `pathlib`. Следующий пример находит все файлы по шаблону `*.txt` в заданном каталоге:

```
import pathlib

for filename in path.Path('dirname').glob('*.txt'):
    print(filename)
```

Если вы используете `rglob()` вместо `glob()`, функция проведет рекурсивный поиск по всем подкаталогам имен файлов, соответствующих шаблону. Обе функции возвращают генератор, производящий имена файлов при переборе.

9.10. ФУНКЦИЯ PRINT()

Для вывода серии значений, разделенных пробелами, передайте их функции `print()`:

```
print('The values are', x, y, z)
```

Для подавления или изменения завершителя строки используется `end`:

```
# Подавление завершающего символа новой строки
print('The values are', x, y, z, end='')
```

Если нужно перенаправить вывод в файл, используйте `file`:

```
# Перенаправление в объект файла f
print('The values are', x, y, z, file=f)
```

Для изменения разделителя элементов используйте `sep`:

```
# Вывод запятых между значениями
print('The values are', x, y, z, sep=',')
```

9.11. ГЕНЕРАЦИЯ ВЫВОДА

Прямые операции с файлами лучше всего знакомы программистам. Но функции-генераторы могут использоваться и для выдачи потока ввода/вывода как последовательности фрагментов данных. Для этого используйте команду `yield`, как бы вы обычно использовали функцию `write()` или `print()`:

```
def countdown(n):
    while n > 0:
        yield f'T-minus {n}\n'
        n -= 1
    yield 'Kaboom!\n'
```

Такая генерация выходного потока обеспечивает гибкость. Она отделена от кода, фактически направляющего поток в приемник. Направить вывод выше в файл `f` можно так:

```
lines = countdown(5)
f.writelines(lines)
```

Перенаправить вывод через сокет `s` можно так:

```
for chunk in lines:
    s.sendall(chunk.encode('utf-8'))
```

Сохранить весь вывод в одной строке можно так:

```
out = ''.join(lines)
```

Более сложные приложения могут использовать этот подход для самостоятельной реализации ввода/вывода. Генератор может выдавать небольшие части текста, а другая функция будет собирать их в большие буферы для выполнения одной, более эффективной операции ввода/вывода.

```
chunks = []
buffered_size = 0
for chunk in count:
    chunks.append(chunk)
    buffered_size += len(chunk)
    if buffered_size >= MAXBUFFERSIZE:
        outf.write(''.join(chunks))
        chunks.clear()
        buffered_size = 0
outf.write(''.join(chunks))
```

Для программ, направляющих вывод в файлы или сетевые подключения, решение с генератором может привести к снижению затрат памяти. Весь выходной поток часто может генерироваться и обрабатываться небольшими частями (без предварительного объединения в одну большую выходную строку или список строк).

9.12. ПОТРЕБЛЕНИЕ ВХОДНЫХ ДАННЫХ

Для программ, потребляющих фрагментарные входные данные, расширенные генераторы могут пригодиться для декодирования протоколов и других аспектов ввода/вывода. Пример расширенного генератора, который получает байтовые фрагменты и собирает их в строки:

```
def line_receiver():
    data = bytearray()
    line = None
    linecount = 0
    while True:
        part = yield line
```



```

linecount += part.count(b'\n')
data.extend(part)
if linecount > 0:
    index = data.index(b'\n')
    line = bytes(data[:index+1])
    data = data[index+1:]
    linecount -= 1
else:
    line = None

```

Здесь генератор программируется для получения байтовых фрагментов, которые объединяются в один массив. Если в нем будет новая строка, она извлекается и возвращается. Иначе возвращается `None`:

```

>>> r = line_receiver()
>>> r.send(None) # Необходимый первый шаг
>>> r.send(b'hello')
>>> r.send(b'world\nit ')
b'hello world\n'
>>> r.send(b'works!')
>>> r.send(b'\n')
b'it works!\n'
>>>

```

У этого подхода есть интересный побочный эффект: он выносит наружу фактические операции ввода/вывода, которые должны выполняться для получения входных данных. А именно, `line_receiver()` вообще не содержит операций ввода/вывода! Значит, она может использоваться в разных контекстах, например с сокетами:

```

r = line_receiver()
data = None
while True:
    while not (line:=r.send(data)):
        data = sock.recv(8192)

    # Обработка строки
    ...

```

или с файлами:

```

r = line_receiver()
data = None
while True:
    while not (line:=r.send(data)):
        data = file.read(10000)

    # Обработка строки
    ...

```

и даже в асинхронном коде:

```
async def reader(ch):
    r = line_receiver()
    data = None
    while True:
        while not (line:=r.send(data)):
            data = await ch.receive(8192)

        # Обработка строки
    ...
```

9.13. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ

Иногда нужно сериализовать представление объекта для передачи по сети, сохранения в файле или в базе данных. В одном из возможных решений данные преобразуются в стандартную кодировку (например, JSON или XML). Есть и популярный формат сериализации данных `pickle`, специфический для Python.

Модуль `pickle` сериализует объект в поток байтов, который можно использовать для восстановления объекта позднее. Интерфейс `pickle` очень прост. Он состоит всего из двух операций: `dump()` и `load()`. Например, следующий код записывает объект в файл:

```
import pickle
obj = SomeObject()
with open(filename, 'wb') as file:
    pickle.dump(obj, file) # Сохранение объекта в f
```

Восстановление объекта выполняется так:

```
with open(filename, 'rb') as file:
    obj = pickle.load(file) # Восстановление объекта
```

Формат данных `pickle` использует свой механизм формирования записей. Так, последовательность объектов может быть сохранена серией операций `dump()`, выполняемых друг за другом. Для восстановления этих объектов используется простая последовательность операций `load()`.

В сетевом программировании `pickle` часто используется для создания сообщений, закодированных в байтовом виде. Для этого используются `dumps()` и `loads()`. Вместо чтения/записи данных в файл эти функции работают с байтовыми строками.

```
obj = SomeObject()

# Преобразование объекта в байты
data = pickle.dumps(obj)
...

# Обратное преобразование байтов в объект
obj = pickle.loads(data)
```

Объектам, определяемым пользователем, обычно не нужно ничего специально делать для работы с `pickle`. Но некоторые их разновидности с `pickle` не работают. Обычно это объекты, включающие состояние времени выполнения: открытые файлы, потоки, замыкания, генераторы и т. д. Для обработки таких случаев класс может определить специальные методы `__getstate__()` и `__setstate__()`.

Метод `__getstate__()` может вызываться для создания значения, представляющего состояние объекта. `__getstate__()` обычно возвращает строку, кортеж, список или словарь. `__setstate__()` получает это значение при восстановлении и должен восстановить состояние объекта по нему.

При кодировании объекта `pickle` не добавляет исходный код в основе — он кодирует ссылку на имя определяющего класса. При восстановлении это имя используется для поиска исходного кода в системе.

Чтобы восстановление успешно работало, у получателя уже должен быть установлен правильный исходный код. Важно, что модуль `pickle` небезопасен по своей природе: распаковка ненадежных данных — это известная лазейка для удаленного выполнения кода. Поэтому `pickle` лучше использовать, только если вы можете полностью обезопасить исполнительную среду.

9.14. БЛОКИРУЮЩИЕ ОПЕРАЦИИ И ПАРАЛЛЕЛИЗМ

Концепция *блокировки* стала одной из основных концепций ввода/вывода. Ввод/вывод, по сути, связан с реальным миром. Часто он подразумевает ожидание готовности ввода или устройств. Например, код, читающий данные из сети, может выполнить операцию получения данных через сокет:

```
data = sock.recv(8192)
```

Во время выполнения команда может немедленно вернуть управление при наличии данных. Но если данных нет, она останавливается в ожидании их

поступления. Это называется *блокировкой*. Пока программа заблокирована, в ней ничего не происходит.

В сценариях анализа данных или простых программах вам не придется беспокоиться о блокировке. Но если вы хотите, чтобы ваша программа занималась чем-то другим во время выполнения блокирующей операции, придется действовать иначе. Возможность выполнения сразу нескольких операций — базовая задача параллелизма. Типичный пример — одновременное чтение из двух и более сетевых сокетов в программе:

```
def reader1(sock):
    while (data := sock.recv(8192)):
        print('reader1 got:', data)

def reader2(sock):
    while (data := sock.recv(8192)):
        print('reader2 got:', data)

# Задача: как обеспечить одновременное
# выполнение reader1() и reader2()?
```

В оставшейся части этого раздела описаны другие подходы к решению этой задачи. Но это не полноценный учебник по теме параллельного выполнения. За подробной информацией обратитесь к другим ресурсам.

9.14.1. Неблокирующий ввод/вывод

Один из подходов к предотвращению блокировки основан на *неблокирующем вводе/выводе*. Это специальный режим, который нужно включить, например, для сокета:

```
sock.setblocking(False)
```

После включения при переходе операции в блокирующее состояние выдается исключение:

```
try:
    data = sock.recv(8192)
except BlockingIOError as e:
    # Доступные данные отсутствуют
    ...
```

В ответ на `BlockingIOError` программа может начать работу над чем-то другим. Она может повторить операцию ввода/вывода позднее, чтобы узнать о появлении данных. Одновременное чтение из двух сокетов может выполняться так:

```
def reader1(sock):
    try:
        data = sock.recv(8192)
        print('reader1 got:', data)
    except BlockingIOError:
        pass

def reader2(sock):
    try:
        data = sock.recv(8192)
        print('reader2 got:', data)
    except BlockingIOError:
        pass

def run(sock1, sock2):
    sock1.setblocking(False)
    sock2.setblocking(False)
    while True:
        reader1(sock1)
        reader2(sock2)
```

На практике постоянное использование неблокирующего ввода/вывода получается громоздким и неэффективным. Основную часть этой программы составляет функция `run()`, которая будет выполняться в неэффективном цикле активного ожидания — она постоянно пытается читать данные из сокетов. Такое решение работает, но эту архитектуру нельзя назвать хорошей.

9.14.2. Опрос каналов ввода/вывода

Можно не полагаться на исключения и активное ожидание, а вместо этого опрашивать каналы ввода/вывода и проверять наличие данных. Для этого используйте модуль `select` или `selectors`.

Ниже приведена слегка измененная версия функции `run()`:

```
from selectors import DefaultSelector, EVENT_READ, EVENT_WRITE

def run(sock1, sock2):
    selector = DefaultSelector()
    selector.register(sock1, EVENT_READ, data=reader1)
    selector.register(sock2, EVENT_READ, data=reader2)
    # Ожидать, пока что-нибудь не произойдет
    while True:
        for key, evt in selector.select():
            func = key.data
            func(key.fileobj)
```

В этом коде цикл передает управление `reader1()` или `reader2()` через обратный вызов при обнаружении ввода/вывода в соответствующем сокете. Сама операция `selector.select()` блокируется, ожидая появления ввода/вывода. В отличие от прошлого примера, программа не будет активно расходовать такты процессора.

Такой подход к вводу/выводу лежит в основе многих «асинхронных» фреймворков (например, `asyncio`). Хотя обычно внутренние механизмы цикла событий остаются скрытыми.

9.14.3. Потoki

В двух последних примерах параллелизм требовал использования специальной функции `run()` для управления вычислениями. Вместо этого можно воспользоваться многопоточным программированием и модулем `threading`.

Программный поток (`thread`) можно рассматривать как независимую задачу внутри вашей программы. Код в следующем примере читает данные из двух сокетов одновременно:

```
import threading

def reader1(sock):
    while (data := sock.recv(8192)):
        print('reader1 got:', data)

def reader2(sock):
    while (data := sock.recv(8192)):
        print('reader2 got:', data)

t1 = threading.Thread(target=reader1, args=[sock1])
t2 = threading.Thread(target=reader2, args=[sock2])

// Запустить потоки
t1.start()
t2.start()

# Ожидать завершения потоков
t1.join()
t2.join()
```

Здесь `reader1()` и `reader2()` выполняются одновременно. Ими управляет операционная система, так что вам почти ничего не нужно знать о принципе их работы. Выполнение блокирующей операции в одном потоке не повлияет на другой. Тема многопоточного программирования очень объемная для нашей книги. Но некоторые дополнительные примеры будут приведены в разделе модуля `threading` позднее.

9.14.4. Параллельное выполнение в `asyncio`

Модуль `asyncio` предоставляет альтернативу потокам реализации параллелизма. Во внутреннем устройстве она основана на цикле событий, использующем опрос каналов ввода/вывода. Но высокоуровневая модель программирования очень похожа на многопоточное решение благодаря использованию специальных асинхронных функций:

```
import asyncio

async def reader1(sock):
    loop = asyncio.get_event_loop()
    while (data := await loop.sock_recv(sock, 8192)):
        print('reader1 got:', data)

async def reader2(sock):
    loop = asyncio.get_event_loop()
    while (data := await loop.sock_recv(sock, 8192)):
        print('reader2 got:', data)

async def main(sock1, sock2):
    loop = asyncio.get_event_loop()
    t1 = loop.create_task(reader1(sock1))
    t2 = loop.create_task(reader2(sock2))

    # Ожидание завершения задач
    await t1
    await t2

...
# Запуск
asyncio.run(main(sock1, sock2))
```

Описание всех функций `asyncio` потребовало бы отдельной книги. Пока нам достаточно знать, что многие библиотеки и фреймворки заявляют о поддержке асинхронных операций. Это значит, что параллельное выполнение поддерживается средствами `asyncio` или похожего модуля. Возможно, в коде будут использоваться асинхронные функции и другие взаимосвязанные средства.

9.15. МОДУЛИ СТАНДАРТНОЙ БИБЛИОТЕКИ

Многие модули стандартной библиотеки используются для разных операций, связанных с вводом/выводом. В этом разделе приводится краткий обзор часто используемых модулей с примерами. Полную справочную информацию можно найти в интернете или в IDE, я не буду повторять ее здесь.

Главная цель этого раздела — показать читателю верное направление. Я поделюсь модулями, которыми вы будете пользоваться, и приведу ряд примеров очень распространенных задач программирования, где задействован каждый из них.

Многие примеры приводятся в виде интерактивных сеансов Python. Это эксперименты, которые вам стоит опробовать самостоятельно.

9.15.1. Модуль `asyncio`

Модуль `asyncio` поддерживает параллельные операции ввода/вывода с применением опроса каналов ввода/вывода и цикла событий. Он предназначен для использования в коде, работающем с сетями и распределенными системами. Пример эхо-сервера TCP с применением низкоуровневых сокетов:

```
import asyncio
from socket import *

async def echo_server(address):
    loop = asyncio.get_event_loop()
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    sock.setblocking(False)
    print('Server listening at', address)
    with sock:
        while True:
            client, addr = await loop.sock_accept(sock)
            print('Connection from', addr)
            loop.create_task(echo_client(loop, client))

async def echo_client(loop, client):
    with client:
        while True:
            data = await loop.sock_recv(client, 10000)
            if not data:
                break
            await loop.sock_sendall(client, b'Got:' + data)
        print('Connection closed')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.create_task(echo_server('', 25000))
    loop.run_forever()
```


Чтобы протестировать этот код, подключитесь к порту 25 000 на вашем компьютере с помощью `nc` или `telnet`. Программа должна повторять вводимый вами текст. Если вы подключитесь одновременно из нескольких окон терминала, то увидите, что программа может обрабатывать все подключения одновременно.

Большинство приложений, использующих `asyncio`, будут работать не на уровне сокетов, а на более высоком. Но в таких приложениях вы все равно будете использовать специальные асинхронные функции и как-то взаимодействовать с циклом событий.

9.15.2. Модуль `binascii`

`binascii` содержит функции для преобразования двоичных данных в текстовые представления — в частности, в шестнадцатеричную запись и `base64`:

```
>>> binascii.b2a_hex(b'hello')
b'68656c6c6f'
>>> binascii.a2b_hex(_)
b'hello'
>>> binascii.b2a_base64(b'hello')
b'aGVsbG8=\n'
>>> binascii.a2b_base64(_)
b'hello'
>>>
```

Сходная функциональность предоставляется модулем `base64` и методами `hex()` и `fromhex()` типа `bytes`:

```
>>> a = b'hello'
>>> a.hex()
'68656c6c6f'
>>> bytes.fromhex(_)
b'hello'
>>> import base64
>>> base64.b64encode(a)
b'aGVsbG8='
>>>
```

9.15.3. Модуль `cgi`

Допустим, вы хотите разместить на своем сайте простую форму, например для подписки на ежедневный бюллетень «Коты и их виды». Конечно, можно установить новейший веб-фреймворк и потратить неделю-другую на его

освоение. А можно написать простой сценарий CGI — решение в духе «старой школы». Модуль `cgi` предназначен именно для этого.

Предположим, веб-страница содержит следующий фрагмент формы:

```
<form method="POST" action="cgi-bin/register.py">
  <p>
    To register, please provide a contact name and email address.
  </p>
  <div>
    <input name="name" type="text">Your name:</input>
  </div>
  <div>
    <input name="email" type="email">Your email:</input>
  </div>
  <div class="modal-footer justify-content-center">
    <input type="submit" name="submit" value="Register"></input>
  </div>
</form>
```

Сценарий CGI, получающий данные формы на другом конце:

```
#!/usr/bin/env python
import cgi
try:
    form = cgi.FieldStorage()
    name = form.getvalue('name')
    email = form.getvalue('email')
    # Проверка ответов и соответствующая реакция
    ...
    # Генерация результата HTML (или перенаправление)
    print("Status: 302 Moved\r")
    print("Location: https://www.mywebsite.com/thanks.html\r")
    print("\r")
except Exception as e:
    print("Status: 501 Error\r")
    print("Content-type: text/plain\r")
    print("\r")
    print("Some kind of error occurred.\r")
```

Обеспечит ли вам такой сценарий CGI работу в интернет-стартапе? Скорее всего, нет. Поможет ли он решить реальную проблему? Наверное, да.

9.15.4. Модуль `configparser`

Формат файлов INI часто используется для кодирования конфигурационных данных в удобной для человека форме:

```
# config.ini
; Комментарий
[section1]
name1 = value1
name2 = value2
[section2]
; Альтернативный синтаксис
name1: value1
name2: value2
```

`configparser` используется для чтения файлов `.ini` и извлечения данных из них:

```
import configparser
# Создание парсера и чтение файла
cfg = configparser.ConfigParser()
cfg.read('config.ini')

# Извлечение значений
a = cfg.get('section1', 'name1')
b = cfg.get('section2', 'name2')
...
```

Доступна и более расширенная функциональность, включающая интерполяцию строк, возможность слияния нескольких файлов `.ini`, определения значений по умолчанию и т. д. За примерами обращайтесь к дополнительным источникам.

9.15.5. Модуль `csv`

`csv` предназначен для чтения/записи CSV-файлов (значений, разделенных запятыми). Формат CSV генерируется такими программами, как Microsoft Excel, или экспортируется из баз данных. Для использования этого модуля откройте файл и наложите на него дополнительный уровень кодирования/декодирования CSV:

```
import csv

# Чтение файла CSV в список кортежей
def read_csv_data(filename):
    with open(filename, newline='') as file:
        rows = csv.reader(file)
        # Чтение заголовка, который часто содержится в первой строке
        headers = next(rows)
        # Теперь прочитайте остальные данные
        for row in rows:
```

```

        # Обработка текущей строки
        ...

# Запись данных Python в файл CSV
def write_csv_data(filename, headers, rows):
    with open(filename, 'w', newline='') as file:
        out = csv.writer(file)
        out.writerow(headers)
        out.writerows(rows)

```

Часто вместо этого используется вспомогательный метод `DictReader()`. Он интерпретирует первую строку CSV-файла как набор заголовков и возвращает каждую строку как словарь вместо кортежа.

```

import csv

def find_nearby(filename):
    with open(filename, newline='') as file:
        rows = csv.DictReader(file)
        for row in rows:
            lat = float(rows['latitude'])
            lon = float(rows['longitude'])
            if close_enough(lat, lon):
                print(row)

```

`csv` не делает с данными CSV почти ничего, кроме их чтения или записи. Главное преимущество этого модуля в том, что он умеет правильно кодировать/декодировать данные и обрабатывает многие граничные случаи, включая данные в кавычках, специальные символы и другие подробности.

Этот модуль может использоваться для написания простых сценариев чистки данных или их подготовки к использованию в других программах. Если вы намерены решать задачи анализа данных с данными CSV, рассмотрите возможность использования стороннего пакета, такого как популярная библиотека `pandas`.

9.15.6. Модуль `errno`

При каждой ошибке системного уровня Python сообщает о ней исключением-субклассом `OSError`. Некоторые распространенные виды системных ошибок представляются отдельными субклассами `OSError`: `PermissionError` или `FileNotFoundError`.

На практике могут встречаться сотни других ошибок. Для них любое исключение `OSError` содержит числовой атрибут `errno`, который может проверяться программой. Модуль `errno` дает символические константы, соответствующие

этим кодам ошибок. Они часто используются при написании специализированных обработчиков исключений. Следующий обработчик ошибок выполняется при нехватке свободного места на устройстве:

```
import errno

def write_data(file, data):
    try:
        file.write(data)
    except OSError as e:
        if e.errno == errno.ENOSPC:
            print("You're out of disk space!")
        else:
            raise # Другая ошибка - распространить
```

9.15.7. Модуль fcntl

fcntl применяется для выполнения низкоуровневых управляющих операций ввода/вывода в UNIX с использованием системных функций fcntl() и ioctl(). Также он полезен при блокировке файлов — эта задача иногда возникает в контексте параллелизма и распределенных систем. Пример открытия файла в сочетании с взаимоисключающей блокировкой по всем процессам с использованием fcntl.flock():

```
import fcntl

with open("somefile", "r") as file:
    try:
        fcntl.flock(file.fileno(), fcntl.LOCK_EX)
        # Использование файла
        ...
    finally:
        fcntl.flock(file.fileno(), fcntl.LOCK_UN)
```

9.15.8. Модуль hashlib

hashlib предоставляет функции для вычисления криптографических хеш-кодов — MD5 и SHA-1:

```
>>> h = hashlib.new('sha256')
>>> h.update(b'Hello') # Feed data
>>> h.update(b'World')
>>> h.digest()
b'\xa5\x91\xa6\xd4\x0b\xf4 @J\x01\x173\xcf\xb7\xb1\x90\xd6,e\xbf\x0b\xcd
\xa3+W\xb2w\xd9\xad\x9f\x14n'
>>> h.hexdigest()
'a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e'
```

```
>>> h.digest_size
32
>>>
```

9.15.9. Пакет http

В `http` содержится большой объем кода, относящегося к низкоуровневой реализации интернет-протокола HTTP. Он может использоваться для реализации как серверов, так и клиентов. Но большая часть пакета считается устаревшей и слишком низкоуровневой для повседневной работы. Серьезный программист, работающий с HTTP, скорее воспользуется такими сторонними библиотеками, как `requests`, `httpx`, `Django`, `flask` и т. д.

Одна из полезных скрытых возможностей пакета `http` — запуск автономного веб-сервера. Введите в каталоге с набором файлов следующую команду:

```
bash $ python -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Теперь Python будет предоставлять файлы вашему браузеру, если тот подключится к правильному порту. Для управления веб-сайтом эта возможность не подойдет, но она пригодится для тестирования и отладки программ, использующих веб-технологии. Например, автор использовал ее для локального тестирования программ, содержащих комбинацию HTML, JavaScript и WebAssembly.

9.15.10. Модуль io

`io` в основном содержит определения классов для реализации объектов файлов, возвращаемых функцией `open()`. Прямые обращения к таким классам встречаются редко. Но модуль содержит и пару классов, которые могут пригодиться для «имитации» файла в форме строк и байтов.

Это может быть удобно для тестирования и других применений, где нужно предоставить файл, но данные на самом деле были получены иначе.

`StringIO()` дает интерфейс «в стиле файла» для строк. Запись вывода в строку может выполняться так:

```
# Функция, которая получает файл
def greeting(file):
    file.write('Hello\n')
    file.write('World\n')

# Функция вызывается с реальным файлом
```

```
with open('out.txt', 'w') as file:
    greeting(file)

# Функция вызывается с "фиктивным" файлом
import io
file = io.StringIO()
greeting(file)

# Получение вывода
output = file.getvalue()
```

Точно так же можно создать объект `StringIO` и использовать его для чтения:

```
file = io.StringIO('hello\nworld\n')
while (line := file.readline()):
    print(line, end='')
```

Класс `BytesIO()` служит похожей цели, но используется для эмуляции двоичного ввода/вывода с байтами.

9.15.11. Модуль `json`

`json` может использоваться для кодирования и декодирования данных в формат JSON, обычно используемый в API микросервисов и веб-приложений. Для преобразования данных определены две основные функции: `dumps()` и `loads()`. `dumps()` получает словарь Python и кодирует его в JSON-строку в «Юникоде»:

```
>>> import json
>>> data = { 'name': 'Mary A. Python', 'email': 'mary123@python.org' }
>>> s = json.dumps(data)
>>> s
'{"name": "Mary A. Python", "email": "mary123@python.org"}'
>>>
```

`loads()` работает наоборот:

```
>>> d = json.loads(s)
>>> d == data
True
>>>
```

`dumps()` и `loads()` поддерживают многочисленные параметры для управления разными аспектами преобразования и взаимодействия с экземплярами классов Python. Эта тема выходит за рамки раздела, но подробную информацию о ней можно найти на официальном сайте.

9.15.12. Модуль logging

`logging` считается стандартом для вывода диагностики программы и отладочных данных в стиле `print`. Он может использоваться для перенаправления вывода в файл журнала и дает множество параметров конфигурации. Стандартная практика написания кода, создающего экземпляр `Logger` и выдающего в него сообщения, выглядит так:

```
import logging
log = logging.getLogger(__name__)

# Функция, использующая logging
def func(args):
    log.debug("A debugging message")
    log.info("An informational message")
    log.warning("A warning message")
    log.error("An error message")
    log.critical("A critical message")

# Конфигурация вывода в журнал (выполняется один раз при запуске)
if __name__ == '__main__':
    logging.basicConfig(
        level=logging.WARNING,
        filename="output.log"
    )
```

Поддерживаются пять уровней журнального вывода, упорядоченных по критичности. При настройке журнальной системы вы задаете уровень, который действует как фильтр: выводятся только сообщения этого и большего уровня критичности. Модуль `logging` дает множество параметров конфигурации, в основном связанных с внутренней обработкой журнальных сообщений.

Обычно вам не нужно ничего знать об этом при написании кода приложения — вы просто используете `debug()`, `info()`, `warning()` и другие методы для заданного экземпляра `Logger`. Любая специальная настройка конфигурации выполняется при запуске программы в определенном месте (например, в функции `main()` или основном блоке кода).

9.15.13. Модуль os

`os` предоставляет переносимый интерфейс для общих функций операционной системы, связанных со средой процесса, файлами, каталогами, разрешениями и т. д. Программный интерфейс придерживается традиций программирования C и таких стандартов, как POSIX.

С практической точки зрения большая часть этого модуля относится к слишком низкому уровню для прямого использования в типичном приложении. Но если вы когда-нибудь столкнетесь с проблемой при выполнении необычных низкоуровневых системных операций (например, при открытии ТТУ), вероятно, вы найдете здесь нужный функционал.

9.15.14. Модуль `os.path`

`os.path` — это устаревший модуль для управления путями и выполнения общих операций в файловой системе. Его функциональность была заменена более новым модулем `pathlib`, но он все еще широко применяется на практике и часто встречается в коде.

Одна из основных проблем, решаемых модулем, — портируемая обработка разделителей пути в UNIX (косая черта `/`) и Windows (обратная косая черта `\`). Такие функции, как `os.path.join()` и `os.path.split()`, часто используются для разделения пути на компоненты и их слияния:

```
>>> filename = '/Users/beazley/Desktop/old/data.csv'
>>> os.path.split()
('/Users/beazley/Desktop/old', 'data.csv')
>>> os.path.join('/Users/beazley/Desktop', 'out.txt')
'/Users/beazley/Desktop/out.txt'
>>>
```

Пример кода, где используются эти функции:

```
import os.path

def clean_line(line):
    # Преобразование строки
    return line.strip().upper() + '\n'

def clean_data(filename):
    dirname, basename = os.path.split()
    newname = os.path.join(dirname, basename+'.clean')
    with open(newname, 'w') as out_f:
        with open(filename, 'r') as in_f:
            for line in in_f:
                out_f.write(clean_line(line))
```

Модуль `os.path` поддерживает функции для выполнения проверок в файловой системе и получения метаданных файлов — `isfile()`, `isdir()`, `getsize()` и т. д. Следующая функция возвращает общий размер в байтах для простого файла или всех файлов в каталоге:

```
import os.path
def compute_usage(filename):
    if os.path.isfile(filename):
        return os.path.getsize(filename)
    elif os.path.isdir(filename):
        return sum(compute_usage(os.path.join(filename, name))
                   for name in os.listdir(filename))
    else:
        raise RuntimeError('Unsupported file kind')
```

9.15.15. Модуль pathlib

pathlib — современный инструмент для портируемых и высокоуровневых операций с путями. Он объединяет множество функций, ориентированных на работу с файлами, в одном месте и использует объектно-ориентированный интерфейс. Основной объект — это класс Path:

```
from pathlib import Path

filename = Path('/Users/beazley/old/data.csv')
```

Экземпляр filename класса Path может использоваться для выполнения разных операций с файлом:

```
>>> filename.name
'data.csv'
>>> filename.parent
Path('/Users/beazley/old')
>>> filename.parent / 'newfile.csv'
Path('/Users/beazley/old/newfile.csv')
>>> filename.parts
('/', 'Users', 'beazley', 'old', 'data.csv')
>>> filename.with_suffix('.csv.clean')
Path('/Users/beazley/old/data.csv.clean')
>>>
```

Экземпляры Path содержат функции для получения метаданных файлов, списка содержимого каталогов и других аналогичных функций. Ниже приведена повторная реализация функции compute_usage() из прошлого раздела:

```
import pathlib

def compute_usage(filename):
    pathname = pathlib.Path(filename)
    if pathname.is_file():
        return pathname.stat().st_size
    elif pathname.is_dir():
        return sum(path.stat().st_size
```

```
        for path in pathname.rglob('*')
            if path.is_file()
                return pathname.stat().st_size
    else:
        raise RuntimeError('Unsupported file kind')
```

9.15.16. Модуль re

re используется для поиска и замены текста по шаблону с использованием регулярных выражений:

```
>>> text = 'Today is 3/27/2018. Tomorrow is 3/28/2018.'
>>> # Поиск всех вхождений даты
>>> import re
>>> re.findall(r'\d+/\d+/\d+', text)
['3/27/2018', '3/28/2018']
>>> # Замена всех вхождений даты текстом
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2018-3-27. Tomorrow is 2018-3-28.'
>>>
```

Регулярные выражения пользуются дурной славой из-за своего сложного синтаксиса. В этом примере `\d+` интерпретируется как «одна или несколько цифр». За дополнительной информацией о синтаксисе шаблонов обращайтесь к официальной документации модуля re.

9.15.17. Модуль shutil

shutil используется для выполнения стандартных операций, которые обычно выполняются в командной оболочке. Это операции копирования и удаления файлов, работы с архивами и т. д. Копирование файла выполняется так:

```
import shutil
shutil.copy(srcfile, dstfile)
```

Перемещение файла:

```
shutil.move(srcfile, dstfile)
```

Копирование дерева каталогов:

```
shutil.copytree(srcdir, dstdir)
```

Удаление дерева каталогов:

```
shutil.rmtree(pathname)
```

Модуль `shutil` часто используется как более безопасная и портируемая альтернатива для прямого выполнения команд оболочки функцией `os.system()`.

9.15.18. Модуль `select`

`select` используется для простого опроса нескольких потоков ввода/вывода. Он позволяет отслеживать набор дескрипторов файлов для обнаружения входящих или получения исходящих данных. Типичное применение модуля:

```
import select

# Набор объектов, представляющих дескрипторы файлов. Это должны быть
# целые числа или объекты с методом fileno().
want_to_read = [ ... ]
want_to_write = [ ... ]
check_exceptions = [ ... ]

# Тайм-аут (или None)
timeout = None

# Опрос каналов ввода/вывода
can_read, can_write, have_exceptions = \
    select.select(want_to_read, want_to_write, check_exceptions,
                 timeout)

# Выполнение операций ввода/вывода
for file in can_read:
    do_read(file)
for file in can_write:
    do_write(file)

# Обработка исключений
for file in have_exceptions:
    handle_exception(file)
```

Здесь создаются три набора дескрипторов файлов: для чтения, записи и исключений. Они передаются `select()` вместе с необязательным тайм-аутом. `select()` возвращает три подмножества переданных аргументов. Они представляют файлы, с которыми может быть выполнена запрашиваемая операция. Например, у файла, возвращаемого `can_read()`, есть ожидающие обработки входящие данные.

`select()` — низкоуровневая системная функция. Обычно она используется для отслеживания системных событий и реализации асинхронных фреймворков ввода/вывода (например, встроенного асинхронного модуля `asyncio`).

В дополнение к `select()` модуль `select` предоставляет функции `poll()`, `epoll()`, `kqueue()` и аналогичные варианты функций, обеспечивающие схожую функциональность. Их доступность зависит от операционной системы.

Модуль `selectors` предоставляет высокоуровневый интерфейс к `select`, который может быть удобен в некоторых контекстах. Пример приводился в разделе 9.14.2.

9.15.19. Модуль `smtplib`

`smtplib` реализует клиентскую сторону SMTP, которая обычно используется для отправки сообщений электронной почты. Чаще всего этот модуль применяется именно в сценариях, отправляющих электронную почту:

```
import smtplib

fromaddr = "someone@some.com"
toaddrs = ["recipient@other.com" ]
amount = 123.45
msg = f"""From: {fromaddr}
Pay {amount} bitcoin or else. We're watching.
"""

server = smtplib.SMTP('localhost')
serv.sendmail(fromaddr, toaddrs, msg)
serv.quit()
```

Есть дополнительный функционал для работы с паролями, аутентификацией и другими аспектами. Но если сценарий выполняется на компьютере с поддержкой электронной почты, приведенный пример справится с задачей.

9.15.20. Модуль `socket`

`socket` дает низкоуровневый доступ к функциям сетевого программирования. Интерфейс построен по образцу стандартного интерфейса сокетов BSD, который обычно ассоциируется с системным программированием на языке C.

Следующий пример показывает, как установить исходящее соединение и получить ответ:

```
from socket import socket, AF_INET, SOCK_STREAM

sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('python.org', 80))
sock.send(b'GET /index.html HTTP/1.0\r\n\r\n')
```

```

parts = []
while True:
    part = sock.recv(10000)
    if not part:
        break
    parts.append(part)
parts = b''.join(parts)
print(parts)

```

Ниже представлен простейший эхо-сервер, принимающий клиентские соединения и возвращающий все полученные данные в эхо-режиме. Для тестирования сервера запустите его и подключитесь командой вида `telnet localhost 25000` или `nc localhost 25000` в отдельном терминальном сеансе.

```

from socket import socket, AF_INET, SOCK_STREAM

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_handler(client, addr)

def echo_handler(client, addr):
    print('Connection from:', addr)
    with client:
        while True:
            data = client.recv(10000)
            if not data:
                break
            client.sendall(data)
    print('Connection closed')

if __name__ == '__main__':
    echo_server(('', 25000))

```

Для серверов UDP процесс соединения не используется, но сервер все равно должен связать сокет с известным адресом. Типичный пример сервера и клиента UDP:

```

# udp.py

from socket import socket, AF_INET, SOCK_DGRAM

def run_server(address):
    sock = socket(AF_INET, SOCK_DGRAM) # 1. Создание сокета UDP
    sock.bind(address)                # 2. Связывание с адресом/портом
    while True:

```

```

    msg, addr = sock.recvfrom(2000) # 3. Получение сообщения
    # ... do something
    response = b'world'
    sock.sendto(response, addr)      # 4. Возвращение ответа

def run_client(address):
    sock = socket(AF_INET, SOCK_DGRAM) # 1. Создание сокета UDP
    sock.sendto(b'hello', address)      # 2. Отправка сообщения
    response, addr = sock.recvfrom(2000) # 3. Получение ответа
    print("Received:", response)
    sock.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 4:
        raise SystemExit('Usage: udp.py [-client|-server] hostname port')
    address = (sys.argv[2], int(sys.argv[3]))
    if sys.argv[1] == '-server':
        run_server(address)
    elif sys.argv[1] == '-client':
        run_client(address)

```

9.15.21. Модуль struct

`struct` используется для преобразования данных между Python и двоичными структурами данных в виде байтовых строк в нем. Такие структуры данных часто используются при взаимодействии с функциями, написанными на C, двоичными форматами файлов, сетевыми протоколами или при передаче двоичных данных через последовательные порты.

Допустим, вы хотите построить двоичное сообщение, формат которого описывается структурой данных C:

```

# Формат сообщения: все значения используют
#                               обратный (big endian) порядок байтов
struct Message {
    unsigned short msgid; // 16-битное целое без знака
    unsigned int sequence; // 32-битный последовательный номер
    float x;               // 32-битное число с плавающей точкой
    float y;               // 32-битное число с плавающей точкой
}

```

С помощью модуля `struct` это можно сделать так:

```

>>> import struct
>>> data = struct.pack('>HIff', 123, 456, 1.23, 4.56)
>>> data
b'\x00{\x00\x00\x00-?\x9dp\xa4@\x91\xeb\x85'
>>>

```

Для декодирования двоичных данных используется вызов `struct.unpack`:

```
>>> struct.unpack('>HIff', data)
(123, 456, 1.2300000190734863, 4.559999942779541)
>>>
```

Различия в значениях с плавающей точкой связаны с потерей точности при их преобразовании в 32-битные значения. В Python значения с плавающей точкой представляются в виде 64-битных с двойной точностью.

9.15.22. Модуль `subprocess`

`subprocess` используется для выполнения отдельной программы как подпроцесса, но с контролем за средой выполнения, включая обработку ввода/вывода, завершение и т. д. У модуля есть два распространенных применения.

Чтобы запустить отдельную программу и получить сразу весь ее вывод, используйте `check_output()`:

```
import subprocess

# Выполнить команду 'netstat -a' и получить весь ее вывод
try:
    out = subprocess.check_output(['netstat', '-a'])
except subprocess.CalledProcessError as e:
    print("It failed:", e)
```

Данные, возвращаемые `check_output()`, отображаются в виде байтов. Чтобы преобразовать их в текст, не забудьте выбрать верную кодировку:

```
text = out.decode('utf-8')
```

Можно настроить канал и взаимодействовать с подпроцессом более детально. Для этого используйте класс `Popen`:

```
import subprocess

# wc - программа, которая возвращает количество строк, слов и байтов
p = subprocess.Popen(['wc'],
                     stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE)

# Отправка данных подпроцессу
p.stdin.write(b'hello world\nthis is a test\n')
p.stdin.close()

# Чтение данных
out = p.stdout.read()
print(out)
```


Экземпляр `Popen` содержит атрибуты `stdin` и `stdout`, которые могут использоваться для обмена данными с подпроцессом.

9.15.23. Модуль `tempfile`

`tempfile` поддерживает создание временных файлов и каталогов. Пример создания временного файла:

```
import tempfile

with tempfile.TemporaryFile() as f:
    f.write(b'Hello World')
    f.seek(0)
    data = f.read()
    print('Got:', data)
```

По умолчанию временные файлы открываются в двоичном режиме и поддерживают и чтение, и запись. Команда `with` часто используется и для определения области видимости, где будет использоваться файл. В конце блока `with` файл будет удален.

Создать временный каталог можно так:

```
with tempfile.TemporaryDirectory() as dirname:
    # Использование каталога dirname
    ...
```

Как и в случае с файлами, каталог со всем его содержимым будет удален в конце блока `with`.

9.15.24. Модуль `textwrap`

`textwrap` может использоваться для форматирования текста под конкретную ширину терминала. Это довольно специализированный модуль, но он часто может использоваться для очистки выводимого текста при построении отчетов. Интерес для нас представляют две функции.

`wrap()` получает текст и переносит его по заданной ширине столбца. Функция возвращает список строк:

```
import textwrap
text = """look into my eyes
look into my eyes
the eyes the eyes the eyes
not around the eyes
```

```

don't look around the eyes
look into my eyes you're under
"""

```

```

wrapped = textwrap.wrap(text, width=81)
print('\n'.join(wrapped))
# Produces:
# look into my eyes look into my eyes the
# eyes the eyes the eyes not around the
# eyes don't look around the eyes look
# into my eyes you're under

```

`indent()` может использоваться для расстановки отступов в блоке текста:

```

print(textwrap.indent(text, ' '))
# Produces:
#   look into my eyes
#   look into my eyes
#   the eyes the eyes the eyes
#   not around the eyes
#   don't look around the eyes
#   look into my eyes you're under

```

9.15.25. Модуль `threading`

`threading` используется для одновременного выполнения кода. Эта задача часто встречается при обработке ввода/вывода в сетевых программах. Многопоточное программирование — обширная тема. Но в следующих примерах продемонстрированы решения некоторых распространенных задач.

Пример запуска потока и ожидания его завершения:

```

import threading
import time

def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(1)

t = threading.Thread(target=countdown, args=[10])
t.start()
t.join()            # Ожидать завершения потока

```

Чтобы не ждать завершения потока, сделайте его демоническим. Для этого нужно передать дополнительный флаг `daemon`:

```

t = threading.Thread(target=countdown, args=[10], daemon=True)

```

Для завершения потока нужно использовать флаг или какую-нибудь специальную переменную. Поток должен быть запрограммирован для проверки состояния этой переменной.

```
import threading
import time

must_stop = False

def countdown(n):
    while n > 0 and not must_stop:
        print('T-minus', n)
        n -= 1
        time.sleep(1)
```

Если потоки собираются изменять общие данные, защитите их с помощью Lock.

```
import threading

class Counter:
    def __init__(self):
        self.value = 0
        self.lock = threading.Lock()

    def increment(self):
        with self.lock:
            self.value += 1

    def decrement(self):
        with self.lock:
            self.value -= 1
```

Если один поток должен ожидать, пока другой поток что-то сделает, используйте Event.

```
import threading
import time

def step1(evt):
    print('Step 1')
    time.sleep(5)
    evt.set()

def step2(evt):
    evt.wait()
    print('Step 2')

evt = threading.Event()
threading.Thread(target=step1, args=[evt]).start()
threading.Thread(target=step2, args=[evt]).start()
```

Если потоки должны взаимодействовать, используйте `Queue`:

```
import threading
import queue
import time

def producer(q):
    for i in range(10):
        print('Producing:', i)
        q.put(i)
    print('Done')
    q.put(None)

def consumer(q):
    while True:
        item = q.get()
        if item is None:
            break
        print('Consuming:', item)
    print('Goodbye')

q = queue.Queue()
threading.Thread(target=producer, args=[q]).start()
threading.Thread(target=consumer, args=[q]).start()
```

9.15.26. Модуль `time`

`time` используется для обращения к системным функциям, относящимся ко времени. Ниже перечислены самые полезные функции:

- `sleep(seconds)` — заставляет Python приостановить выполнение на интервал времени в секундах, заданный числом с плавающей точкой.
- `time()` — возвращает текущее системное время в формате UTC в виде числа с плавающей точкой. Время измеряется в секундах от начала эпохи (обычно 1 января 1970 г. для систем UNIX). Используйте `localtime()` для преобразования значения в структуру данных, удобную для извлечения полезной информации.
- `localtime([secs])` — возвращает объект `struct_time`, отображающий местное время в системе или время, представленное значением с плавающей точкой `secs`, которое передается в аргументе. Полученная структура содержит атрибуты `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min`, `tm_sec`, `tm_wday`, `tm_yday` и `tm_isdst`.
- `gmtime([secs])` — то же, что `localtime()`, только итоговая структура отображает время в формате UTC (или GMT — Greenwich Mean Time).

- `ctime([secs])` — преобразует время в секундах в текстовую строку для вывода. Функция хорошо подходит для отладки и ведения журнала.
- `asctime(tm)` — преобразует структуру времени, представленную `localtime()`, в текстовую строку для вывода.

`datetime` обычно используется с целью представления даты и времени для выполнения вычислений, связанных с датой/временем, и работы с часовыми поясами.

9.15.27. Пакет `urllib`

`urllib` используется для выдачи запросов HTTP на стороне клиента. Самая полезная функция — `urllib.request.urlopen()`. Она может использоваться для получения простых веб-страниц:

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://www.python.org')
>>> data = u.read()
>>>
```

Для кодировки параметров формы используйте функцию `urllib.parse.urlencode()`:

```
from urllib.parse import urlencode
from urllib.request import urlopen

form = {
    'name': 'Mary A. Python',
    'email': 'mary123@python.org'
}

data = urlencode(form)
u = urlopen('http://httpbin.org/post', data.encode('utf-8'))
response = u.read()
```

`urlopen()` хорошо подходит для простых веб-страниц и API, где задействован протокол HTTP или HTTPS. Но эта функция становится менее удобной, если в обращениях к странице задействованы cookie, расширенные схемы аутентификации и другие уровни. Большинство программистов Python в таких ситуациях воспользуются сторонней библиотекой (`requests` или `httpx`). Вам стоит сделать то же самое.

Подпакет `urllib.parse` содержит дополнительные функции для управления самими URL-адресами. Например, `urlparse()` может использоваться для разбора URL:

```
>>> url = 'http://httpbin.org/get?name=Dave&n=42'
>>> from urllib.parse import urlparse
>>> urlparse(url)
ParseResult(scheme='http', netloc='httpbin.org', path='/get', params='',
query='name=Dave&n=42', fragment='')
>>>
```

9.15.28. Модуль unicodedata

`unicodedata` используется для более сложных операций, где участвуют строки текста в «Юникоде». Часто один текст в «Юникоде» может иметь несколько представлений.

Например, символ U+00F1 (ñ) может быть полностью сформирован как одиночный символ U+00F1 или же разложен в многосимвольную последовательность U+006e U+0303 (n, ~). Это может создать проблемы в программах, предполагающих, что текстовые строки, визуально отображающие одно и то же, на самом деле будут такими же в представлении. Возьмем следующий пример с ключами словаря:

```
>>> d = {}
>>> d['Jalape\x1f1o'] = 'spicy'
>>> d['Jalapen\u0303o'] = 'mild'
>>> d
{'jalapeño': 'spicy', 'jalapeño': 'mild' }
>>>
```

Сначала это кажется очевидной ошибкой: как может словарь иметь два идентичных, но при этом разных ключа? Дело в том, что ключи состоят из разных символьных последовательностей «Юникода».

При появлении проблем с логической целостностью обработки строк «Юникода», одинаково выглядящих на экране, их нужно нормализовать. Функция `unicodedata.normalize()` обеспечивает однозначное представление символов. Вызов `unicodedata.normalize('NFC', s)` гарантирует, что все символы в строке `s` полностью сформированы и не представляются комбинациями составляющих символов. Вызов `unicodedata.normalize('NFD', s)` гарантирует, что все символы в `s` полностью разложены на составляющие.

Модуль `unicodedata` содержит функции для проверки свойств символов — регистра и принадлежности к категориям (цифры, пропуски и т. д.). Общие свойства символов проверяются функцией `unicodedata.category(c)`. `unicodedata.category('A')` возвращает значение `'Lu'`, означающее, что символ служит буквой верхнего регистра. Дополнительную информацию об этих

значениях можно найти в официальной базе данных символов «Юникода»: <https://www.unicode.org/ucd>.

9.15.29. Пакет xml

Пакет `xml` содержит большую подборку модулей для обработки данных XML. Но прочитать документ XML и извлечь информацию из него проще всего с помощью подпакета `xml.etree`. Допустим, у вас есть документ XML в файле `recipe.xml`, который выглядит так:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
  <title>Famous Guacamole</title>
  <description>A southwest favorite!</description>
  <ingredients>
    <item num="4"> Large avocados, chopped </item>
    <item num="1"> Tomato, chopped </item>
    <item num="1/2" units="C"> White onion, chopped </item>
    <item num="2" units="tbl"> Fresh squeezed lemon juice </item>
    <item num="1"> Jalapeno pepper, diced </item>
    <item num="1" units="tbl"> Fresh cilantro, minced </item>
    <item num="1" units="tbl"> Garlic, minced </item>
    <item num="3" units="tsp"> Salt </item>
    <item num="12" units="bottles"> Ice-cold beer </item>
  </ingredients>
  <directions>
    Combine all ingredients and hand whisk to desired consistency.
    Serve and enjoy with ice-cold beers.
  </directions>
</recipe>
```

Следующая программа извлекает отдельные элементы из него:

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
title = doc.find('title')
print(title.text)

# Альтернатива (только текст элемента)
print(doc.findtext('description'))

# Перебор нескольких элементов
for item in doc.findall('ingredients/item'):
    num = item.get('num')
    units = item.get('units', '')
    text = item.text.strip()
    print(f'{num} {units} {text}')
```

9.16. НАПОСЛЕДОК

Ввод/вывод — важнейшая часть любой полезной программы. Из-за своей популярности Python может работать почти с любыми форматами данных, кодировками и структурами документов. Если что-то не поддерживается стандартной библиотекой, вы почти наверняка найдете сторонний модуль для решения вашей задачи.

В общей перспективе полезнее думать о границах вашего приложения. На внешней границе между программой и окружающим миром часто встречаются проблемы, связанные с кодировкой данных. В первую очередь это относится к текстовым данным и «Юникоду». Большая часть сложности обработки ввода/вывода в Python (поддержка разных кодировок, политика обработки ошибок и т. д.) связана именно с этой проблемой. Очень важно помнить, что текстовые данные отделяются от двоичных. Четкое понимание того, с чем работаете, поможет вам увидеть общую картину.

Второй важный фактор ввода/вывода — общая модель вычислений. Сейчас код Python разделен на две области — простой синхронный и асинхронный код, обычно ассоциируемый с модулем `asyncio` (для него характерно использование функций `async` и синтаксис `async/await`).

Асинхронный код почти всегда требует использования специальных библиотек, способных работать в этой среде. Это заставляет вас писать код приложения в «асинхронном» стиле. По возможности избегайте асинхронного программирования, если только вы твердо не уверены в том, что без него не обойтись, а если не уверены, то почти наверняка можно обойтись. Большинство качественных программ в Python написаны в обычном синхронном стиле. Он создает намного меньше проблем с анализом, отладкой и тестированием.

ГЛАВА 10

Встроенные функции и стандартная библиотека

Эта глава — компактный справочник по встроенным функциям Python. Они всегда доступны в программе без каких-либо команд `import`. Глава завершается краткой сводкой нескольких полезных модулей стандартной библиотеки.

10.1. ВСТРОЕННЫЕ ФУНКЦИИ

- `abs(x)` — возвращает модуль (абсолютное значение) `x`.
- `all(s)` — возвращает `True`, если все значения в итерируемом объекте `s` интерпретируются как `True`. Возвращает `True`, если объект `s` пуст.
- `any(s)` — возвращает `True`, если хотя бы одно значение в итерируемом объекте `s` интерпретируется как `True`. Возвращает `True`, если объект `s` пуст.
- `ascii(x)` — создает представление объекта `x` для вывода, как и `repr()`, но использует только ASCII-символы. Те, что не входят в ASCII, преобразуются в соответствующие служебные последовательности. Функция может использоваться для просмотра строк в «Юникоде» на терминале или командной оболочке без поддержки «Юникода».
- `bin(x)` — возвращает строку с двоичным представлением целого числа `x`.
- `bool([x])` — тип, представляющий логические значения `True` и `False`. В случае использования `x` для преобразования, возвращает `True`, если `x` интерпретируется как истинное значение в обычной семантике проверки истинности — ненулевое число, непустой список и т. д. В остальных случаях возвращается `False`. Значение `False` возвращается по умолчанию и при вызове `bool()` без аргументов. Класс `bool`

унаследован от `int`, так что логические значения `True` и `False` могут использоваться как целые числа со значениями 1 и 0 в математических вычислениях.

- `breakpoint()` — устанавливает точку останова отладчика вручную. При обнаружении управление передается `pdb`, отладчику Python.
- `bytearray([x])` — представляет изменяемый массив байтов. При создании экземпляра `x` может быть итерируемой последовательностью целых чисел от 0 до 255, восьмибитной строкой, байтовым литералом или целым числом, задающим размер байтового массива (в этом случае каждый элемент инициализируется нулем).
- `bytearray(s, encoding)` — альтернативная схема вызова для создания экземпляра `bytearray` из символов строки `s`. Аргумент `encoding` задает кодировку символов, которая должна использоваться при преобразовании.
- `bytes([x])` — тип, представляющий неизменяемый массив байтов.
- `bytes(s, encoding)` — альтернативная схема вызова для создания экземпляра `bytes` из символов строки `s`. Аргумент `encoding` задает кодировку символов, которая должна использоваться при преобразовании.

В табл. 10.1 перечислены операции, поддерживаемые `bytes` и `bytearray`.

Таблица 10.1. Операции с байтовыми строками и массивами

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--|--|
| <code>s + t</code> | Конкатенация, если <code>t</code> — байтовая строка |
| <code>s * n</code> | Дублирование, если <code>n</code> — целое число |
| <code>s % x</code> | Форматирует байты, <code>x</code> — кортеж |
| <code>s[i]</code> | Возвращает элемент <code>i</code> в виде целого числа |
| <code>s[i:j]</code> | Возвращает сегмент |
| <code>s[i:j:stride]</code> | Возвращает расширенный сегмент |
| <code>len(s)</code> | Количество байтов в <code>s</code> |
| <code>s.capitalize()</code> | Делает первый символ заглавным |
| <code>s.center(width [, pad])</code> | Вывравнивает строку по центру поля длины <code>width</code> , <code>pad</code> — символ заполнения |
| <code>s.count(sub [, start [, end]])</code> | Подсчитывает вхождения заданной подстроки <code>sub</code> |
| <code>s.decode([encoding [, errors]])</code> | Декодирует байтовую строку в текст (только для типа <code>bytes</code>) |

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|---|
| <code>s.endswith(suffix [, start [, end]])</code> | Проверяет, что строка завершается заданным суффиксом |
| <code>s.expandtabs([tabsize])</code> | Заменяет табуляции пробелами |
| <code>s.find(sub [, start [, end]])</code> | Находит первое вхождение заданной подстроки <code>sub</code> |
| <code>s.hex()</code> | Преобразует в шестнадцатеричную строку |
| <code>s.index(sub [, start [, end]])</code> | Находит первое вхождение заданной подстроки <code>sub</code> или выдает ошибку |
| <code>s.isalnum()</code> | Проверяет, являются ли все символы алфавитно-цифровыми |
| <code>s.isalpha()</code> | Проверяет, являются ли все символы алфавитными |
| <code>s.isascii()</code> | Проверяет, принадлежат ли все символы кодировке ASCII |
| <code>s.isdigit()</code> | Проверяет, являются ли все символы цифрами |
| <code>s.islower()</code> | Проверяет, относятся ли все символы к нижнему регистру |
| <code>s.isspace()</code> | Проверяет, являются ли все символы пропусками |
| <code>s.istitle()</code> | Проверяет, что первая буква каждого слова в строке относится к верхнему регистру |
| <code>s.isupper()</code> | Проверяет, относятся ли все символы к верхнему регистру |
| <code>s.join(t)</code> | Объединяет последовательность строк <code>t</code> с ограничителем <code>s</code> |
| <code>s.ljust(width [, fill])</code> | Выравнивает строку <code>s</code> по левому краю в строке размера <code>width</code> |
| <code>s.lower()</code> | Преобразует в нижний регистр |
| <code>s.lstrip([chrs])</code> | Удаляет начальные пропуски или символы в <code>chrs</code> |
| <code>s.maketrans(x [, y [, z]])</code> | Создает таблицу преобразования для <code>s.translate()</code> |
| <code>s.partition(sep)</code> | Разделяет строку на основе строки-разделителя <code>sep</code> . Возвращает кортеж (<code>head</code> , <code>sep</code> , <code>tail</code>) или (<code>s</code> , <code>''</code> , <code>''</code>), если разделитель <code>sep</code> не найден |
| <code>s.removeprefix(prefix)</code> | Возвращает <code>s</code> без заданного префикса <code>prefix</code> , если он есть |
| <code>s.removesuffix(suffix)</code> | Возвращает <code>s</code> без заданного суффикса <code>suffix</code> , если он есть |

Таблица 10.1 (окончание)

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|--|
| <code>s.replace(old, new [, maxreplace])</code> | Заменяет подстроку |
| <code>s.rfind(sub [, start [, end]])</code> | Находит последнее вхождение подстроки |
| <code>s.rindex(sub [, start [, end]])</code> | Находит последнее вхождение подстроки или выдает ошибку |
| <code>s.rjust(width [, fill])</code> | Выводит строку <code>s</code> по правому краю в строке размера <code>width</code> |
| <code>s.rpartition(sep)</code> | Разделяет строку на основе строки-разделителя <code>sep</code> , но поиск проводится от конца строки |
| <code>s.rsplit([sep [, maxsplit]])</code> | Разбивает строку с конца по разделителю <code>sep</code> . <code>maxsplit</code> – максимальное число выполняемых разбиений. Если аргумент <code>maxsplit</code> опущен, результат идентичен методу <code>split()</code> |
| <code>s.rstrip([chrs])</code> | Удаляет завершающие пропуски или символы в <code>chrs</code> |
| <code>s.split([sep [, maxsplit]])</code> | Разбивает строку с конца по разделителю <code>sep</code> . <code>maxsplit</code> – максимальное число выполняемых разбиений |
| <code>s.splitlines([keepends])</code> | Разделяет строку на список внутренних строк. Если аргумент <code>keepends</code> равен 1, завершающиеся символы конца строки сохраняются |
| <code>s.startswith(prefix [, start [, end]])</code> | Проверяет, что строка начинается с заданного префикса |
| <code>s.strip([chrs])</code> | Удаляет начальные и завершающие пропуски или символы в <code>chrs</code> |
| <code>s.swapcase()</code> | Преобразует верхний регистр в нижний и наоборот |
| <code>s.title()</code> | Возвращает версию строки, где все слова начинаются с букв верхнего регистра |
| <code>s.translate(table [, deletechars])</code> | Преобразует строку по таблице <code>table</code> , с удалением символов из <code>deletechars</code> |
| <code>s.upper()</code> | Преобразует строку к верхнему регистру |
| <code>s.zfill(width)</code> | Дополняет строку нулями слева до заданной ширины |

Байтовые массивы дополнительно поддерживают операции из табл. 10.2.

Таблица 10.2. Дополнительные операции с байтовыми массивами

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--------------------------------|---|
| <code>s[i] = v</code> | Присваивание элементу |
| <code>s[i:j] = t</code> | Сегментное присваивание |
| <code>s[i:j:stride] = t</code> | Расширенное сегментное присваивание |
| <code>del s[i]</code> | Удаление элемента |
| <code>del s[i:j]</code> | Удаление сегмента |
| <code>del s[i:j:stride]</code> | Расширенное удаление сегмента |
| <code>s.append(x)</code> | Присоединяет новый байт в конец |
| <code>s.clear()</code> | Очищает байтовый массив |
| <code>s.copy()</code> | Создает копию |
| <code>s.extend(t)</code> | Расширяет <code>s</code> байтами из <code>t</code> |
| <code>s.insert(n, x)</code> | Вставляет байт <code>x</code> в позицию с индексом <code>n</code> |
| <code>s.pop([n])</code> | Удаляет и возвращает байт с индексом <code>n</code> |
| <code>s.remove(x)</code> | Удаляет первое вхождение байта <code>x</code> |
| <code>s.reverse()</code> | Выполняет обратную перестановку байтового массива на месте |

- `callable(obj)` — возвращает `True`, если `obj` может вызываться как функция.
- `chr(x)` — преобразует целое число `x`, представляющее кодовый пункт «Юникода», в односимвольную строку.
- `classmethod(func)` — декоратор создает метод класса для функции `func`. Обычно он используется только внутри определений классов, где он неявно вызывается с помощью `@classmethod`. В отличие от обычных, метод класса получает в первом аргументе класс, а не экземпляр.
- `compile(string, filename, kind)` — компилирует строку в объект кода для использования с `exec()` или `eval()`. Аргумент `string` — строка с действительным кодом Python. Если код состоит из нескольких строк, они должны завершаться одиночным символом новой строки (`'\n'`), а не платформенно-зависимыми разновидностями (как `'\r\n'` в Windows). Строка `filename` содержит имя файла, где была определена строка (при наличии). Аргумент `kind` содержит `'exec'` для последовательности команд, `'eval'` для одного выражения или `'single'` для одной исполняемой команды. Возвращаемый объект кода можно напрямую передать `exec()` или `eval()` вместо строки.

- `complex([real[, imag]])` — тип, представляющий комплексное число с вещественной и мнимой частью `real` и `imag`. Если аргумент `imag` не указан, мнимой части присваивается 0. Если `real` передается в виде строки, строка разбирается и преобразуется в комплексное число. Тогда аргумент `imag` должен быть опущен. Если `real` содержит любую другую разновидность объекта, возвращается значение `real.__complex__()`. Если аргументы не заданы, возвращается `0j`.

В табл. 10.3 представлены методы и атрибуты `complex`.

Таблица 10.3. Атрибуты `complex`

| АТРИБУТ/МЕТОД | ОПИСАНИЕ |
|----------------------------|---|
| <code>z.real</code> | Вещественная часть |
| <code>z.imag</code> | Мнимая часть |
| <code>z.conjugate()</code> | Вычисляет сопряженное комплексное число |

- `delattr(object, attr)` — удаляет атрибут объекта. Аргумент `attr` содержит строку. То же, что `del object.attr`.
- `dict([m])` или `dict(key1=value1, key2=value2, ...)` — тип, представляющий словарь. Если аргумент не задан, возвращается пустой словарь. Если `m` — объект отображения (например, другим словарем), возвращается новый словарь с такими же ключами и значениями. Например, если `m` является словарем, `dict(m)` создает его поверхностную копию. Если объект `m` не служит отображением, он должен поддерживать перебор, генерирующий последовательность пар (ключ, значение). Пары используются для заполнения словаря. `dict()` может вызываться с ключевыми аргументами. Например, `dict(foo=3, bar=7)` создает словарь `{'foo': 3, 'bar': 7}`.

В табл. 10.4 перечислены операции, поддерживаемые словарями.

Таблица 10.4. Операции словарей

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|-----------------------|---|
| <code>m n</code> | Объединяет <code>m</code> и <code>n</code> в один словарь |
| <code>len(m)</code> | Возвращает число элементов в <code>m</code> |
| <code>m[k]</code> | Возвращает элемент <code>m</code> с ключом <code>k</code> |
| <code>m[k]=x</code> | Присваивает <code>m[k]</code> значение <code>x</code> |
| <code>del m[k]</code> | Удаляет <code>m[k]</code> из <code>m</code> |

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--------------------------------------|---|
| <code>k in m</code> | Возвращает <code>True</code> , если <code>k</code> – ключ в <code>m</code> |
| <code>m.clear()</code> | Удаляет все элементы из <code>m</code> |
| <code>m.copy()</code> | Создает поверхностную копию <code>m</code> |
| <code>m.fromkeys(s [, value])</code> | Создает новый словарь с ключами из последовательности <code>s</code> . Со всеми ключами связывается значение <code>value</code> |
| <code>m.get(k [, v])</code> | Возвращает элемент <code>m[k]</code> , если он найден. Иначе возвращается <code>v</code> |
| <code>m.items()</code> | Возвращает пары «ключ – значение» |
| <code>m.keys()</code> | Возвращает ключи |
| <code>m.pop(k [, default])</code> | Возвращает элемент <code>m[k]</code> , если он найден, и удаляет его из <code>m</code> . Иначе возвращается значение <code>default</code> , если оно передано, или <code>KeyError</code> в противном случае |
| <code>m.popitem()</code> | Удаляет из <code>m</code> случайную пару (ключ, значение) и возвращает ее в виде кортежа |
| <code>m.setdefault(k [, v])</code> | Возвращает элемент <code>m[k]</code> , если он найден. Иначе возвращает <code>v</code> и присваивает <code>m[k]=v</code> |
| <code>m.update(b)</code> | Добавляет все объекты из <code>b</code> в <code>m</code> |
| <code>m.values()</code> | Возвращает значения |

- `dir([object])` — возвращает отсортированный список имен атрибутов. Если `object` — модуль, результат содержит список всех символических имен, определенных в нем. Если `object` — тип или объект класса, возвращается список имен атрибутов. Имена обычно берутся из атрибута `__dict__` объекта, если он определен, но могут использоваться и другие источники. Если аргумент не задан, возвращаются имена из текущей таблицы локальных символических имен. Заметьте, что функция прежде всего используется для получения информации (например, в интерактивном режиме в командной строке). Лучше не использовать ее для формального анализа программ: полученная информация может быть неполной. Кроме того, определенные пользователем классы могут определять специальный метод `__dir__()`, изменяющий результат этой функции.
- `divmod(a, b)` — возвращает частное и остаток от деления в виде кортежа. Для целых чисел возвращается значение `(a // b, a % b)`. Для чисел с плавающей точкой возвращается `(math.floor(a / b), a % b)`. Эта функция не может вызываться для комплексных чисел.

- `enumerate(iter, start=0)` — для итерируемого объекта `iter` возвращает новый итератор (типа `enumerate`), производящий кортежи со счетчиком и значением, полученным от `iter`. Например, если `iter` производит значения `a, b, c`, то `enumerate(iter)` произведет `(0, a), (1, b), (2, c)`. Необязательный аргумент `start` изменяет начальное значение счетчика.
- `eval(expr [, globals [, locals]])` — вычисляет выражение. `expr` — строка или объект кода, созданный вызовом `compile()`. `globals` и `locals` — объекты отображений, определяющие глобальное и локальное пространства имен для операции. Если эти аргументы не заданы, то выражение вычисляется со значениями, полученными при вызовах `globals()` и `locals()` в среде вызывающей стороны.

Чаще всего `globals` и `locals` определяются как словари, но более сложные приложения могут использовать нестандартные объекты отображений.

- `exec(code [, globals [, locals]])` — выполняет команды Python. Аргумент `code` содержит строку, байты или объект кода, созданный вызовом `compile()`. `globals` и `locals` определяют глобальное и локальное пространства имен для операции. Если эти аргументы не заданы, то выражение вычисляется со значениями, полученными при вызовах `globals()` и `locals()` в среде вызывающей стороны.
- `filter(function, iterable)` — создает итератор, возвращающий элементы из `iterable`, для которых `function()` возвращает `True`.
- `float([x])` — тип, представляющий число с плавающей точкой. Если `x` — число, оно преобразуется в число с плавающей точкой. Если `x` — строка, она преобразуется в число с плавающей точкой. Для всех остальных объектов вызывается `x.__float__()`. Если аргумент не задан, возвращается `0.0`.

В табл. 10.5 показаны методы и атрибуты чисел с плавающей точкой.

Таблица 10.5. Методы и атрибуты чисел с плавающей точкой

| АТРИБУТ/МЕТОД | ОПИСАНИЕ |
|-----------------------------------|--|
| <code>x.real</code> | Вещественная часть при использовании в качестве комплексного числа |
| <code>x.imag</code> | Мнимая часть при использовании в качестве комплексного числа |
| <code>x.conjugate()</code> | Вычисляет сопряженное комплексное число |
| <code>x.as_integer_ratio()</code> | Преобразует в пару «числитель/знаменатель» |

| АТТРИБУТ/МЕТОД | ОПИСАНИЕ |
|-------------------------------|--|
| <code>x.hex()</code> | Создает шестнадцатеричное представление |
| <code>x.is_integer()</code> | Проверяет, является ли число точным целым значением |
| <code>float.fromhex(s)</code> | Создает из шестнадцатеричной строки. Является методом класса |

- `format(value [, format_spec])` — преобразует значение в отформатированную строку в соответствии со строкой форматной спецификации, заданной в `format_spec`. Эта операция вызывает метод `value.__format__()`, который может интерпретировать форматную спецификацию как считает нужным.

Для простых типов данных спецификатор обычно состоит из символа выравнивания ('<', '>' или '^'), числа (задающего ширину поля) и символьного кода 'd', 'f' или 's' для целого числа, числа с плавающей точкой или строкового значения. Например, форматная спецификация 'd' форматирует целое число, спецификация '8d' выравнивает его по правому краю в восьмисимвольном поле, а '<8d' — по левому краю там же. За дополнительной информацией о `format()` и спецификаторах обращайтесь к главе 9.

- `frozenset([items])` — тип, представляющий объект неизменяемого множества, заполненный значениями из `items`, который должен быть итерируемым. Значения тоже должны быть неизменяемыми. Если аргумент не задан, возвращается пустое множество. `frozenset` поддерживает все операции для множеств, кроме операций, изменяющих множество на месте.
- `getattr(object, name [, default])` — возвращает значение именованного атрибута объекта `object`. Строка `name` содержит имя атрибута. `default` — необязательное значение, возвращаемое, если атрибута не существует. Иначе выдается ошибка `AttributeError`. Равнозначно обращению `object.name`.
- `globals()` — возвращает словарь, представляющий глобальное пространство имен для текущего модуля. При вызове из другой функции или метода возвращает глобальное пространство имен модуля, где была определена функция или метод.
- `hasattr(object, name)` — возвращает `True`, если `name` — имя атрибута объекта `object.name` — является строкой.
- `hash(object)` — возвращает целочисленный хеш-код для объекта (по возможности). Хеш-код в основном используется в реализации

словарей, множеств и других объектов отображений. Хеш-код всегда одинаков для любых объектов, которые сравниваются как равные. Изменяемые объекты обычно не определяют хеш-код, хотя определяемые пользователем классы могут определять метод `__hash__()` для поддержки этой операции.

- `hex(x)` — создает шестнадцатеричную строку из целого числа `x`.
- `id(object)` — возвращает уникальный целочисленный идентификатор объекта. Возвращаемое значение не нужно никак интерпретировать (это не адрес памяти).
- `input([prompt])` — выводит в стандартный вывод приглашение и читает одну входную строку из стандартного ввода. Возвращаемая строка никак не изменяется и не содержит завершитель строки (например, `'\n'`).
- `int(x[, base])` — тип, представляющий целое число. Если `x` — число, оно преобразуется в целое число округлением в сторону 0. Если `x` — строка, она разбирается в целое число. Необязательный аргумент `base` задает основание системы счисления при преобразовании строки.

Помимо поддержки обычных математических операций, целые числа также имеют ряд атрибутов и методов (табл. 10.6).

Таблица 10.6. Методы и атрибуты целых чисел

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--|--|
| <code>x.numerator</code> | Числитель при представлении числа в виде дроби |
| <code>x.denominator</code> | Знаменатель при представлении числа в виде дроби |
| <code>x.real</code> | Вещественная часть при использовании в качестве комплексного числа |
| <code>x.imag</code> | Мнимая часть при использовании в качестве комплексного числа |
| <code>x.conjugate()</code> | Вычисляет сопряженное комплексное число |
| <code>x.bit_length()</code> | Число битов для представления значения в двоичном виде |
| <code>x.to_bytes(length, byteorder, *, signed=False)</code> | Преобразует в байты |
| <code>int.from_bytes(bytes, byteorder, *, signed=False)</code> | Преобразует из байтов. Является методом класса |

- `isinstance(object, classobj)` — возвращает `True`, если `object` — экземпляр `classobj` или subclasses `classobj`. Параметр `classobj` может содержать кортеж возможных типов или классов. Например, `isinstance(s, (list, tuple))` возвращает `True`, если `s` — кортеж или список.
- `issubclass(class1, class2)` — возвращает `True`, если `class1` — subclass `class2`. `class2` может содержать кортеж возможных классов. В этом случае будут проверены все классы. Результат `issubclass(A, A)` равен `True`.
- `iter(object [, sentinel])` — возвращает итератор для генерации элементов из объекта `object`. Если параметр `sentinel` опущен, объект должен либо предоставить метод `__iter__()`, создающий итератор, либо реализовать метод `__getitem__()`, получающий целочисленные аргументы, начиная с 0.

Если параметр `sentinel` задан, `object` интерпретируется иначе. Это должен быть вызываемый (callable) объект, не получающий параметров. Возвращаемый объект-итератор будет многократно вызывать эту функцию, пока возвращаемое значение не будет равно `sentinel`. Тогда перебор останавливается. Если `object` не поддерживает перебор, генерируется ошибка `TypeError`.

- `len(s)` — возвращает число элементов в объекте `s`, который должен быть той или иной разновидностью контейнера (список, кортеж, строка, множество или словарь).
- `list([items])` — тип, представляющий список. `items` может быть любым итерируемым объектом, значения которого используются для заполнения списка. Если `items` уже является списком, создается поверхностная копия. Если аргумент не задан, возвращается пустой список.

В табл. 10.7 представлены операции, определенные для списков.

Таблица 10.7. Операции и методы списков

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|----------------------------|---|
| <code>s + t</code> | Конкатенация, если <code>t</code> — список |
| <code>s * n</code> | Дублирование, если <code>n</code> — целое число |
| <code>s[i]</code> | Возвращает элемент <code>i</code> из <code>s</code> |
| <code>s[i:j]</code> | Возвращает сегмент |
| <code>s[i:j:stride]</code> | Возвращает расширенный сегмент |
| <code>s[i] = v</code> | Присваивание элементу |

Таблица 10.7 (окончание)

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--|---|
| <code>s[i:j] = t</code> | Присваивание сегмента |
| <code>s[i:j:stride] = t</code> | Расширенное присваивание сегмента |
| <code>del s[i]</code> | Удаление элемента |
| <code>del s[i:j]</code> | Удаление сегмента |
| <code>del s[i:j:stride]</code> | Расширенное удаление сегмента |
| <code>len(s)</code> | Количество элементов в <code>s</code> |
| <code>s.append(x)</code> | Присоединяет новый элемент <code>x</code> в конец <code>s</code> |
| <code>s.extend(t)</code> | Присоединяет новый список <code>t</code> в конец <code>s</code> |
| <code>s.count(x)</code> | Подсчитывает вхождения <code>x</code> в <code>s</code> |
| <code>s.index(x [, start [, stop]])</code> | Возвращает наименьшее <code>i</code> , где <code>s[i] == x</code> . Необязательные аргументы <code>start</code> и <code>stop</code> задают начальный и конечный индекс для поиска |
| <code>s.insert(i, x)</code> | Вставляет <code>x</code> в позицию с индексом <code>i</code> |
| <code>s.pop([i])</code> | Возвращает элемент <code>i</code> и удаляет его из списка. Если значение <code>i</code> не задано, возвращается последний элемент |
| <code>s.remove(x)</code> | Ищет <code>x</code> и удаляет его из <code>s</code> |
| <code>s.reverse()</code> | Переставляет элементы <code>s</code> в обратном порядке |
| <code>s.sort([key [, reverse]])</code> | Сортирует элементы <code>s</code> на месте. <code>key</code> – ключевая функция, <code>reverse</code> – флаг сортировки списка в обратном порядке. Значения <code>key</code> и <code>reverse</code> всегда должны задаваться как ключевые аргументы |

- `locals()` — возвращает словарь, дающий локальное пространство имен для стороны вызова. Этот словарь должен использоваться только для анализа исполнительской среды — вносимые в него изменения никак не влияют на соответствующие локальные переменные.
- `map(function, items, ...)` — создает итератор, генерирующий результаты применения `function` к элементам из `items`. Если задано несколько входных последовательностей, предполагается, что функция получает соответствующее число аргументов. При этом все аргументы берутся из разных последовательностей. В этом случае длина результата равна длине кратчайшей входной последовательности.
- `max(s [, args, ...], *, default=obj, key=func)` — для одного аргумента `s` эта функция возвращает максимальное значение среди элементов `s`, который может быть любым итерируемым объектом. Для нескольких

аргументов она возвращает наибольший из своих. Если задан аргумент `default`, который может быть только ключевым, он определяет значение, которое должно возвращаться для пустого `s`. Если задан аргумент `key`, который может быть только ключевым, возвращается значение `v`, для которого `key(v)` возвращает максимальное значение.

- `min(s [, args, ...], *, default=obj, key=func)` — то же, что `max(s)`, но возвращается наименьшее значение.
- `next(s [, default])` — возвращает следующий элемент из итератора `s`. Если у итератора не осталось элементов, выдается исключение `StopIteration`, если только в аргументе `default` не было задано значение по умолчанию. Тогда возвращается `default`.
- `object()` — базовый класс для всех объектов в Python. Его можно вызвать для создания экземпляра, но результат не особо важен.
- `oct(x)` — преобразует целое число `x` в восьмеричную строку.
- `open(filename [, mode [, bufsize [, encoding [, errors [, newline[, closefd]]]]])` — открывает файл `filename` и возвращает объект файла. Аргументы подробно описаны в главе 9.
- `ord(c)` — возвращает целочисленный код одного символа `c`. Это значение обычно соответствует значению кодового пункта символа в «Юникоде».
- `pow(x, y [, z])` — возвращает `x ** y`. Если указан аргумент `z`, эта функция возвращает `(x ** y) % z`. Если заданы все три аргумента, они должны быть целыми числами, а значение `y` должно быть неотрицательным.
- `print(value, ..., *, sep=separator, end=ending, file=outfile)` — выводит набор значений. На вход подается любое число значений, которые все выводятся в одной строке. Ключевой аргумент `sep` позволяет назначить другой символ-разделитель (пробел по умолчанию). `end` задает другой завершитель строки (`'\n'` по умолчанию). `file` перенаправляет вывод в объект файла.
- `property([fget [, fset [, fdel [, doc]]])` — создает атрибут-свойство для классов. `fget` — функция, возвращающая значение атрибута. `fset` задает значение атрибута, а `fdel` удаляет его. `doc` предоставляет строку документации. Свойства часто задаются в виде декоратора:

```
class SomeClass:
    x = property(doc='This is property x')
    @x.getter
    def x(self):
        print('getting x')
```

```

    @x.setter
    def x(self, value):
        print('setting x to', value)

    @x.deleter
    def x(self):
        print('deleting x')

```

- `range([start,] stop [, step])` — создает объект диапазона, представляющий диапазон целых чисел от `start` до `stop`. Аргумент `step` определяет приращение. Если он не указан, используется значение 1. Если значение `start` не указано (когда `range()` вызывается с одним аргументом), по умолчанию используется 0. С отрицательным приращением создается список чисел в порядке убывания.
- `repr(object)` — возвращает строковое представление объекта. В большинстве случаев возвращаемая строка — это выражение, которое может быть передано `eval()` для воссоздания объекта.
- `reversed(s)` — создает обратный итератор для последовательности `s`. Эта функция работает, только если `s` определяет метод `__reversed__()` или реализует методы последовательностей `__len__()` и `__getitem__()`. Не работает с генераторами.
- `round(x [, n])` — округляет число с плавающей точкой `x` до ближайшего кратного 10 в степени минус `n`. Если значение `n` не указано, по умолчанию используется 0. Если два кратных находятся на одинаковом расстоянии, используется округление к 0 в случае, если предыдущая цифра четная, или от 0 в противном случае (например, 0.5 округляется до 0.0, а 1.5 округляется до 2).
- `set([items])` — создает множество, заполненное элементами из итерируемого объекта `items`. Значение `items` должно быть неизменяемым. Если `items` содержит другие множества, они должны относиться к типу `frozenset`. Если аргумента `items` нет, возвращается пустое множество.

В табл. 10.8 представлены операции со множествами.

Таблица 10.8. Операции и методы множеств

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|------------------------|-------------|
| <code>s t</code> | Объединение |
| <code>s & t</code> | Пересечение |
| <code>s - t</code> | Разность |

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|---|
| <code>s ^ t</code> | Симметричная разность |
| <code>len(s)</code> | Возвращает количество элементов в <code>s</code> |
| <code>s.add(item)</code> | Добавляет <code>item</code> в <code>s</code> . Если <code>item</code> уже есть в <code>s</code> , ничего не происходит |
| <code>s.clear()</code> | Удаляет все элементы из <code>s</code> |
| <code>s.copy()</code> | Создает копию <code>s</code> |
| <code>s.difference(t)</code> | Разность множеств. Возвращает все элементы, входящие в <code>s</code> , но не в <code>t</code> |
| <code>s.difference_update(t)</code> | Удаляет из <code>s</code> все элементы, присутствующие в <code>t</code> |
| <code>s.discard(item)</code> | Удаляет элемент <code>item</code> из <code>s</code> . Если <code>item</code> нет в <code>s</code> , ничего не происходит |
| <code>s.intersection(t)</code> | Пересечение. Возвращает все элементы, входящие как в <code>s</code> , так и в <code>t</code> |
| <code>s.intersection_update(t)</code> | Вычисляет пересечение <code>s</code> и <code>t</code> , результат остается в <code>s</code> |
| <code>s.isdisjoint(t)</code> | Возвращает <code>True</code> , если в <code>s</code> и <code>t</code> нет ни одного общего элемента |
| <code>s.issubset(t)</code> | Возвращает <code>True</code> , если <code>s</code> — подмножество <code>t</code> |
| <code>s.issuperset(t)</code> | Возвращает <code>True</code> , если <code>s</code> — надмножество <code>t</code> |
| <code>s.pop()</code> | Возвращает произвольный элемент множества и удаляет его из <code>s</code> |
| <code>s.remove(item)</code> | Удаляет <code>item</code> из <code>s</code> . Если <code>item</code> нет в множестве, выдается ошибка <code>KeyError</code> |
| <code>s.symmetric_difference(t)</code> | Симметричная разность. Возвращает все элементы, входящие в <code>s</code> и <code>t</code> , но не в оба множества |
| <code>s.symmetric_difference_update(t)</code> | Вычисляет симметричную разность <code>s</code> и <code>t</code> , результат остается в <code>s</code> |
| <code>s.union(t)</code> | Объединение. Возвращает все элементы, входящие в <code>s</code> или <code>t</code> |
| <code>s.update(t)</code> | Добавляет все элементы из <code>t</code> в <code>s</code> . <code>t</code> может быть другим множеством, последовательностью или любым объектом, поддерживающим перебор |

- `setattr(object, name, value)` — удаляет атрибут объекта. Аргумент `name` содержит строку. То же, что `object.name = value`.
- `slice([start,] stop [, step])` — возвращает объект сегмента, представляющий целые числа в заданном диапазоне. Объекты сегментов генерируются в расширенном синтаксисе сегментов `a[i:i:k]`.

- `sorted(iterable, *, key=keyfunc, reverse=reverseflag)` — создает отсортированный список из элементов в итерируемом объекте `iterable`. `key` задает функцию с одним аргументом, преобразующую значения перед сравнением. `reverse` содержит флаг сортировки списка в обратном порядке. Аргументы `key` и `reverse` всегда должны задаваться как ключевые, например `sorted(a, key=get_name)`.
- `staticmethod(func)` — создает статический метод для использования в классах. Функция обычно используется как декоратор `@staticmethod`.
- `str([object])` — тип, представляющий строку. Если `object` задан, то строковое представление его значения создается вызовом его метода `__str__()`. Это та же строка, которая отображается при выводе объекта. Если аргумент не задан, создается пустая строка.

В табл. 10.9 представлены методы, определенные для строк.

Таблица 10.9. Операторы и методы строк

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|---|
| <code>s + t</code> | Выполняет конкатенацию, если <code>t</code> — строка |
| <code>s * n</code> | Выполняет дублирование, если <code>n</code> — целое число |
| <code>s % x</code> | Форматирует строку, <code>x</code> — кортеж |
| <code>s[i]</code> | Возвращает элемент <code>i</code> строки |
| <code>s[i:j]</code> | Возвращает сегмент |
| <code>s[i:j:stride]</code> | Возвращает расширенный сегмент |
| <code>len(s)</code> | Количество элементов в <code>s</code> |
| <code>s.capitalize()</code> | Преобразует первый символ к верхнему регистру |
| <code>s.casefold()</code> | Преобразует <code>s</code> в строку, пригодную для сравнения без учета регистра |
| <code>s.center(width [, pad])</code> | Выравнивает строку по центру поля длины <code>width</code> , <code>pad</code> — символ заполнения |
| <code>s.count(sub [, start [, end]])</code> | Подсчитывает вхождения заданной подстроки <code>sub</code> |
| <code>s.decode([encoding [, errors]])</code> | Декодирует байтовую строку в текст (только для типа <code>bytes</code>) |
| <code>s.encode([encoding [, errors]])</code> | Возвращает закодированную версию строки (только для типа <code>str</code>) |
| <code>s.endswith(suffix [, start [, end]])</code> | Проверяет, что строка завершается заданным суффиксом |
| <code>s.expandtabs([tabsize])</code> | Заменяет табуляции пробелами |

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|--|
| <code>s.find(sub [, start [, end]])</code> | Находит первое вхождение заданной подстроки <code>sub</code> |
| <code>s.format(*args, **kwargs)</code> | Форматирует <code>s</code> (только для типа <code>str</code>) |
| <code>s.format_map(m)</code> | Форматирует <code>s</code> с заменами из отображения <code>m</code> (только для типа <code>str</code>) |
| <code>s.index(sub [, start [, end]])</code> | Находит первое вхождение заданной подстроки <code>sub</code> или выдает ошибку |
| <code>s.isalnum()</code> | Проверяет, являются ли все символы алфавитно-цифровыми |
| <code>s.isalpha()</code> | Проверяет, являются ли все символы алфавитными |
| <code>s.isascii()</code> | Проверяет, принадлежат ли все символы кодировке ASCII |
| <code>s.isdecimal()</code> | Проверяет, являются ли все символы цифрами. Не находит совпадения в верхних/нижних индексах и других специальных цифрах |
| <code>s.isdigit()</code> | Проверяет, являются ли все символы цифрами. Находит совпадения в верхних/нижних индексах, но не в простых дробях |
| <code>s.isidentifier()</code> | Проверяет, является ли <code>s</code> допустимым идентификатором Python |
| <code>s.islower()</code> | Проверяет, относятся ли все символы к нижнему регистру |
| <code>s.isnumeric()</code> | Проверяет, являются ли все символы числовыми. Находит совпадения во всех числовых символах, включая простые дроби, римские числа и т. д. |
| <code>s.isprintable()</code> | Проверяет, являются ли символы пригодными для вывода |
| <code>s.isspace()</code> | Проверяет, являются ли все символы пропусками |
| <code>s.istitle()</code> | Проверяет, что первая буква каждого слова в строке относится к верхнему регистру |
| <code>s.isupper()</code> | Проверяет, относятся ли все символы к верхнему регистру |
| <code>s.join(t)</code> | Объединяет последовательность строк <code>t</code> с ограничителем <code>s</code> |
| <code>s.ljust(width [, fill])</code> | Выравнивает строку <code>s</code> по левому краю в строке размера <code>width</code> |
| <code>s.lower()</code> | Преобразует к нижнему регистру |
| <code>s.lstrip([chrs])</code> | Удаляет начальные пропуски или символы в <code>chrs</code> |

Таблица 10.9 (продолжение)

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|---|---|
| <code>s.maketrans(x [, y [, z]])</code> | Создает таблицу преобразования для <code>s.translate()</code> |
| <code>s.partition(sep)</code> | Разделяет строку по разделителю, заданному аргументом <code>sep</code> . Возвращает кортеж (<code>head</code> , <code>sep</code> , <code>tail</code>) или (<code>s</code> , <code>''</code> , <code>''</code>), если разделитель <code>sep</code> не найден |
| <code>s.removeprefix(prefix)</code> | Возвращает <code>s</code> без заданного префикса <code>prefix</code> , если он есть |
| <code>s.removesuffix(suffix)</code> | Возвращает <code>s</code> без заданного суффикса <code>suffix</code> , если он есть |
| <code>s.replace(old, new [, maxreplace])</code> | Заменяет подстроку |
| <code>s.rfind(sub [, start [, end]])</code> | Находит последнее вхождение подстроки |
| <code>s.rindex(sub [, start [, end]])</code> | Находит последнее вхождение подстроки или выдает ошибку |
| <code>s.rjust(width [, fill])</code> | Выравнивает строку <code>s</code> по правому краю в строке размера <code>width</code> |
| <code>s.rpartition(sep)</code> | Разделяет строку по разделителю <code>sep</code> , но поиск проводится от конца строки |
| <code>s.rsplit([sep [, maxsplit]])</code> | Разбивает строку с конца по разделителю <code>sep</code> . <code>maxsplit</code> – максимальное число выполняемых разбиений. Если аргумент <code>maxsplit</code> опущен, результат идентичен методу <code>split()</code> |
| <code>s.rstrip([chars])</code> | Удаляет завершающие пропуски или символы в <code>chars</code> |
| <code>s.split([sep [, maxsplit]])</code> | Разбивает строку с конца по разделителю <code>sep</code> . <code>maxsplit</code> – максимальное число выполняемых разбиений |
| <code>s.splitlines([keepends])</code> | Разделяет строку на список внутренних строк. Если аргумент <code>keepends</code> равен 1, завершающиеся символы конца строки сохраняются |
| <code>s.startswith(prefix [, start [, end]])</code> | Проверяет, что строка начинается с заданного префикса |
| <code>s.strip([chars])</code> | Удаляет начальные и завершающие пропуски или символы в <code>chars</code> |
| <code>s.swapcase()</code> | Преобразует верхний регистр к нижнему, и наоборот |
| <code>s.title()</code> | Возвращает версию строки, где все слова начинаются с букв верхнего регистра |

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--|--|
| <code>s.translate(table [, deletchars])</code> | Преобразует строку по таблице <code>table</code> , с удалением символов из <code>deletchars</code> |
| <code>s.upper()</code> | Преобразует строку к верхнему регистру |
| <code>s.zfill(width)</code> | Дополняет строку нулями слева до заданной ширины |

- `sum(items [, initial])` — вычисляет сумму последовательности элементов из итерируемого объекта `items`. `initial` определяет начальное значение, по умолчанию используется `0`. Обычно эта функция работает только с числами.
- `super()` — возвращает объект, представляющий коллективные супер-классы того класса, где используется функция. Главное предназначение этого объекта — вызов методов базовых классов:

```
class B(A):
    def foo(self):
        super().foo() # Вызвать реализацию foo(),
                     # определенную в суперклассах
```

- `tuple([items])` — тип, представляющий кортеж. Если аргумент `items` задан, он определяет итерируемый объект, который используется для заполнения кортежа. Если же `items` уже является кортежем, он возвращается неизменным. Если аргумент не задан, возвращается пустой кортеж.

В табл. 10.10 представлены методы для кортежей.

Таблица 10.10. Операторы и методы кортежей

| ОПЕРАЦИЯ | ОПИСАНИЕ |
|--|---|
| <code>s + t</code> | Конкатенация, если <code>t</code> — список |
| <code>s * n</code> | Дублирование, если <code>n</code> — целое число |
| <code>s[i]</code> | Возвращает элемент <code>i</code> из <code>s</code> |
| <code>s[i:j]</code> | Возвращает сегмент |
| <code>s[i:j:stride]</code> | Возвращает расширенный сегмент |
| <code>len(s)</code> | Количество элементов в <code>s</code> |
| <code>s.append(x)</code> | Присоединяет новый элемент <code>x</code> в конец <code>s</code> |
| <code>s.count(x)</code> | Подсчитывает вхождения <code>x</code> в <code>s</code> |
| <code>s.index(x [, start [, stop]])</code> | Возвращает наименьшее <code>i</code> , где <code>s[i] == x</code> . Необязательные аргументы <code>start</code> и <code>stop</code> задают начальный и конечный индекс для поиска |

- `type(object)` — базовый класс для всех типов Python. При вызове в качестве функции возвращает `object`. Этот тип совпадает с классом `object`. Для часто используемых типов (целых чисел, чисел с плавающей точкой и списков) тип будет относиться к одному из встроенных классов (`int`, `float`, `list` и т. д.). Для объектов, определяемых пользователем, тип соответствует классу. Для объектов, относящихся к внутренней реализации Python, вы обычно будете получать ссылку на один из классов, определенных в модуле `types`.
- `vars([object])` — возвращает таблицу символических имен объекта (обычно хранящуюся в атрибуте `__dict__`). Если аргумент не задан, возвращается словарь, соответствующий локальному пространству имен. Словарь, возвращаемый этой функцией, должен быть доступен только для чтения. Изменять его содержимое небезопасно.
- `zip([s1 [, s2 [, ...]]])` — создает итератор, производящий кортежи, содержащие по одному элементу из `s1`, `s2` и т. д. `n`-й кортеж состоит из элементов `(s1[n], s2[n], ...)`. Полученный итератор останавливается при исчерпании самого короткого ввода. Если аргументы не заданы, итератор не производит значения.

10.2. ВСТРОЕННЫЕ ИСКЛЮЧЕНИЯ

Здесь описаны встроенные исключения для передачи информации о разных видах ошибок.

10.2.1. Базовые классы исключений

Следующие исключения служат базовыми классами для всех остальных:

- `BaseException` — корневой класс для всех исключений. Все встроенные исключения — производные от этого класса.
- `Exception` — базовый класс для всех исключений, связанных с программой. К этой категории относятся все встроенные исключения, кроме `SystemExit`, `GeneratorExit` и `KeyboardInterrupt`. Исключения, определяемые пользователем, должны быть унаследованы от `Exception`.
- `ArithmeticError` — базовый класс для арифметических исключений, включая `OverflowError`, `ZeroDivisionError` и `FloatingPointError`.
- `LookupError` — базовый класс для ошибок индексирования и обращения по ключу, включая `IndexError` и `KeyError`.

- `EnvironmentError` — базовый класс для ошибок за пределами Python. Синоним для `OSError`.

Эти исключения никогда не выдаются программами явно. Но они могут использоваться для перехвата некоторых классов ошибок. Например, следующий код перехватывает любые ошибки при вычислениях:

```
try:
    # Некоторая операция
    ...
except ArithmeticError as e:
    # Математическая ошибка
```

10.2.2. Атрибуты исключений

Экземпляры исключения `e` содержат ряд стандартных атрибутов для просмотра и/или изменения исключения в некоторых ситуациях.

- `e.args` — кортеж аргументов, переданных при выдаче исключения. Часто кортеж состоит из одного элемента — строки с описанием ошибки. Для исключений `EnvironmentError` значение представляет собой кортеж из 2–3 элементов с целочисленным кодом ошибки, строкой сообщения об ошибке и необязательным именем файла. Содержимое кортежа может пригодиться, если вы хотите заново создать исключение в другом контексте — например, выдать исключение в другом процессе интерпретатора Python.
- `e.__cause__` — предыдущее исключение для явных цепочек исключений.
- `e.__context__` — предыдущее исключение для неявных цепочек исключений.
- `e.__traceback__` — объект трассировки, связанный с исключением.

10.2.3. Предварительно определенные классы исключений

Следующие исключения выдаются программами:

- `AssertionError` — нарушение условия `assert`.
- `AttributeError` — неудачное обращение к атрибуту или присваивание.
- `BufferError` — ожидается буфер в памяти.
- `EOFError` — конец файла; исключение генерируется встроенными функциями `input()` и `raw_input()`. Заметьте, что многие другие

операции ввода/вывода (методы `read()` и `readline()`) файлов для обозначения конца файла возвращают пустую строку, а не выдают исключение.

- **FloatingPointError** — неудачная попытка выполнения операции с плавающей точкой. Помните, что обработка исключений с плавающей точкой — непростая задача, и исключение выдается, только если такая возможность была включена при настройке и сборке Python. Чаще ошибки с плавающей точкой приводят к выдаче таких результатов, как `float('nan')` или `float('inf')`. Субкласс **ArithmeticError**.
- **GeneratorExit** — выдается внутри функции-генератора для обозначения завершения. Это происходит при преждевременном уничтожении генератора (до того, как все его значения были потреблены) или при вызове метода `close()` генератора. Если генератор игнорирует исключение, он завершается, а исключение незаметно игнорируется.
- **IOError** — неудачная операция ввода/вывода. Значение — это экземпляр **IOError** с атрибутами `errno`, `strerror` и `filename`. `errno` содержит целочисленный код ошибки, `strerror` — строковое сообщение об ошибке, а `filename` — необязательное имя файла. Субкласс **EnvironmentError**.
- **ImportError** — возникает, когда оператор `import` не может найти модуль или `from` не может найти имя в модуле.
- **IndentationError** — ошибка отступов. Субкласс **SyntaxError**.
- **IndexError** — индекс последовательности выходит за границы диапазона. Субкласс **LookupError**.
- **KeyError** — ключ не найден в отображении. Субкласс **LookupError**.
- **KeyboardInterrupt** — выдается при нажатии пользователем клавиш прерывания (обычно `Ctrl+C`).
- **MemoryError** — восстанавливаемая ошибка нехватки памяти.
- **ModuleNotFoundError** — команда `import` не находит модуль.
- **NameError** — имя не найдено в локальном или глобальном пространстве имен.
- **NotImplementedError** — нереализованная функциональность. Может выдаваться базовыми классами, требующими, чтобы производные классы реализовали некоторые методы. Субкласс **RuntimeError**.
- **OSError** — ошибка операционной системы. В основном она выдается функциями модуля `os`. Базовый класс для следующих

исключений: `BlockingIOError`, `BrokenPipeError`, `ChildProcessError`, `ConnectionAbortedError`, `ConnectionError`, `ConnectionRefusedError`, `ConnectionResetError`, `FileExistsError`, `FileNotFoundError`, `InterruptedError`, `IsADirectoryError`, `NotADirectoryError`, `PermissionError`, `ProcessLookupError`, `TimeoutError`.

- **OverflowError** — целочисленное значение слишком велико для представления. Обычно это исключение встречается только при передаче больших целочисленных значений объектам, которые во внутренней реализации зависят от целых чисел с фиксированной точностью. Эта ошибка может возникать при работе с объектами `range` или `xrange`, если задать начальные или конечные значения, по размеру превышающие 32 бита. Субкласс `ArithmeticError`.
- **RecursionError** — превышен лимит рекурсии.
- **ReferenceError** — результат обращения по слабой ссылке после уничтожения объекта (см. описание модуля `weakref`).
- **RuntimeError** — общая ошибка, не принадлежащая к другим категориям.
- **StopIteration** — выдается для обозначения конца перебора. Обычно это происходит в методе `next()` объекта или функции-генератора.
- **StopAsyncIteration** — выдается для обозначения конца асинхронного перебора. Исключение применимо только в контексте асинхронных функций и генераторов.
- **SyntaxError** — синтаксическая ошибка парсера. Экземпляры содержат атрибуты `filename`, `lineno`, `offset` и `text`, которые можно использовать для сбора дополнительной информации.
- **SystemError** — внутренняя ошибка интерпретатора. Значение представляет собой строку с описанием проблемы.
- **SystemExit** — выдается функцией `sys.exit()`. Значение представляет собой целое число с кодом возврата. Для немедленного выхода можно воспользоваться вызовом `os._exit()`.
- **TabError** — непоследовательное использование отступов. Генерируется при запуске Python с ключом `-tt`. Субкласс `SyntaxError`.
- **TypeError** — происходит при применении операции или функции к объекту неподходящего типа.
- **UnboundLocalError** — обращение к несвязанной локальной переменной. Ошибка происходит при обращении к переменной до того, как она была определена в функции. Субкласс `NameError`.

- `UnicodeError` — ошибка кодирования или декодирования «Юникода». Субкласс `ValueError`. Базовый класс для следующих исключений: `UnicodeEncodeError`, `UnicodeDecodeError`, `UnicodeTranslateError`.
- `ValueError` — генерируется, когда у аргумента функции или операции правильный тип, но неподходящее значение.
- `WindowsError` — генерируется при неудачных вызовах системных функций в Windows. Субкласс `OSError`.
- `ZeroDivisionError` — деление на ноль. Субкласс `ArithmeticError`.

10.3. СТАНДАРТНАЯ БИБЛИОТЕКА

Python поставляется с крупной стандартной библиотекой. Многие из этих модулей уже были описаны. Справочный материал можно найти по адресу <https://docs.python.org/library>. В этой книге такой информации нет.

Модули ниже заслуживают внимания. Они обычно приносят пользу в широком спектре применений и в программировании Python в целом.

10.3.1. Модуль `collections`

`collections` дополняет Python разными объектами-контейнерами, которые могут пригодиться при работе с данными, например двусторонней очередью (`deque`), словарями, автоматически инициализирующими отсутствующие элементы (`defaultdict`), и счетчиками для табличного представления данных (`Counter`).

10.3.2. Модуль `datetime`

В модуле `datetime` собраны функции, относящиеся к работе с датой, временем и вычислениям с этими данными.

10.3.3. Модуль `itertools`

`itertools` дает много полезных паттернов перебора: сцепление итерируемых объектов, перебор по произведениям множеств, перестановки, группировку и т. д.

10.3.4. Модуль `inspect`

`inspect` предоставляет функции для анализа внутреннего строения элементов, связанных с кодом. Он обычно используется в метапрограммировании функциями, определяющими декораторы и подобные функции.

10.3.5. Модуль **math**

`math` предоставляет распространенные математические функции: `sqrt()`, `cos()`, `sin()` и т. д.

10.3.6. Модуль **os**

В модуле `os` находятся низкоуровневые функции, относящиеся к операционной системе: процессы, файлы, каналы, разрешения и т. д.

10.3.7. Модуль **random**

`random` предоставляет разные функции, связанные с генерацией случайных чисел.

10.3.8. Модуль **re**

`re` поддерживает работу с текстом на базе регулярных выражений.

10.3.9. Модуль **shutil**

`shutil` содержит функции для выполнения типичных операций, обычно применяемых в командной строке (копирования файлов и каталогов).

10.3.10. Модуль **statistics**

`statistics` предоставляет функции для вычисления стандартных статистических характеристик: среднего, медианы, стандартного отклонения и т. д.

10.3.11. Модуль **sys**

`sys` содержит разнообразные атрибуты и методы, относящиеся к исполнительной среде самого Python. Это параметры командной строки, стандартные потоки ввода/вывода, путь импортирования и т. д.

10.3.12. Модуль **time**

В модуле `time` находятся разные функции, относящиеся к системному времени: получение показаний системных часов, паузы и получение затрат процессорного времени в секундах.

10.3.13. Модуль turtle

«Черепашья» графика... Ну, знаете, для детей.

10.3.14. Модуль unittest

`unittest` предоставляет встроенную поддержку для написания модульных тестов. Сам Python тестируется с помощью `unittest`. Но многие программисты предпочитают использовать для тестирования сторонние библиотеки, например `pytest`. И я с ними согласен.

10.4. НАПОСЛЕДОК: ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ МОДУЛЕЙ

На сегодняшний день существует огромное множество пакетов Python, и программисты могут легко искать решения небольших задач в виде зависимостей от сторонних пакетов. Но в Python уже давно есть очень полезный набор встроенных функций и типов данных. Если объединить их с модулями из стандартной библиотеки, для широкого спектра типичных задач программирования часто не нужно ничего сверх того, что уже есть в Python.

Дэвид Бизли

Python. Исчерпывающее руководство

Перевел с английского *Е. Матвеев*

Руководитель дивизиона
Ведущий редактор
Литературный редактор
Художественный редактор
Корректоры
Верстка

*Ю. Сергиенко
Н. Гринчик
Т. Сажина
В. Мостипан
М. Лауконен, М. Молчанова
Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:
194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214,
тел./факс: 208 80 01.

Подписано в печать 19.08.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 1700. Заказ 0000.

Эл Свейгарт

PYTHON. ЧИСТЫЙ КОД ДЛЯ ПРОДОЛЖАЮЩИХ



Вы прошли обучающий курс программирования на Python или прочли несколько книг для начинающих. Что дальше? Как подняться над базовым уровнем, превратиться в крутого разработчика?

«Python. Чистый код для продолжающих» — это не набор полезных советов и подсказок по написанию чистого кода. Вы узнаете о командной строке и других инструментах профессионального разработчика: средствах форматирования кода, статических анализаторах и контроле версий. Вы научитесь настраивать среду разработки, давать имена переменным и функциям, делающие код удобочитаемым, грамотно комментировать и документировать ПО, оценивать быстродействие программ и сложность алгоритмов, познакомитесь с ООП. Такие навыки поднимут вашу ценность как программиста не только в Python, но и в любом другом языке.

Ни одна книга не заменит реального опыта работы и не превратит вас из новичка в профессионала. Но «Чистый код для продолжающих» проведет вас чуть дальше по этому пути: вы научитесь создавать чистый, грамотный, читабельный, легко отлаживаемый код, который можно будет назвать истинно питоническим.

КУПИТЬ