
правильный способ изучить
PYTHON
с примерами и задачами
Климович А.Д.

Климович А.Д.

правильный способ изучить PYTHON с примерами и задачами



Издательство АСТ
г. Москва

УДК 004.38
ББК 32.973.2
К49

Климович, Александр Дмитриевич.

К49 Правильный способ изучить PYTHON с примерами и задачами / А. Д. Климович. — Москва : Издательство АСТ, 2026. — 448 с. — (Программирование от А до С++).

ISBN 978-5-17-178635-9.

Перед вами практическое руководство по программированию на одном из самых востребованных сегодня языков — Python, которое формирует не просто знание синтаксиса, а системное мышление разработчика, способного решать сложные задачи в любой области.

Книга строится на принципе глубокого понимания вместо механического запоминания. Каждая концепция раскрывается через трехмерную модель: что представляет собой конструкция, почему она работает именно так и когда ее следует применять. От первых команд и типов данных до объектно-ориентированного программирования и работы с современными технологиями — читатель проходит путь от обычного пользователя компьютера до создателя реальных программных воплощений.

Особое внимание уделено культуре качественного кода: правильные практики именования, структурирования, документирования и тестирования интегрированы в каждый раздел с самого начала. Книга развивает алгоритмическое мышление — умение декомпозировать сложные проблемы, создавать абстракции и находить элегантные решения.

Автор рассматривает Python не как самоцель, а как инструмент формирования вычислительного мышления, которое применимо далеко за пределами сферы программирования. В книге также затрагиваются этические аспекты разработки и ответственность программиста в создании инклюзивного и безопасного программного продукта.

Данное издание предназначено для всех, кто хочет освоить программирование на качественном уровне — от абсолютных новичков до тех, кто стремится систематизировать свои знания и развить профессиональное мышление разработчика.

УДК 004.38
ББК 32.973.2

ISBN 978-5-17-178635-9

© Оформление. ООО «Интеджер», 2025
© ООО «Издательство АСТ», 2026

Содержание

Предисловие **13**
Добро пожаловать в мир программирования 13
Философия изучения программирования 13
Программирование как способ мышления 13
Понимание вместо запоминания 13
Культура качественного кода 14
Почему Python — идеальный первый язык 14
Философия простоты и читаемости 14
Мультипарадигмальность как преимущество 15
Обширная экосистема и практическое применение 15
Плавная кривая обучения 15
Как правильно использовать книгу 16
Структура изложения материала 16
Рекомендации по изучению 16
Работа с ошибками как инструмент обучения 17
Использование дополнительных ресурсов 17
Настройка среды разработки 18
Выбор интерпретатора Python 18
Установка Python в различных операционных системах 18
Выбор среды разработки 18
Настройка системы управления пакетами 19
Создание виртуального окружения 19
Проверка установки 20

Глава 1. Первые шаги в мире программирования **21**
1.1. Что такое программирование и зачем оно нужно 21
1.2. Python — философия простоты и элегантности 21
1.3. Интерпретатор как ваш первый собеседник 22
1.4. Анатомия ошибок: учимся понимать компьютер 23
1.5. Первая программа: традиция `"Hello, World!"` по-русски 25
1.6. Интерактивный режим vs написание скриптов 27
1.7. Практические упражнения и размышления 29
Упражнение 1. Персонализированное приветствие 29
Упражнение 2. Простой калькулятор с проверкой 29
Упражнение 3. Анализатор числовых последовательностей 30
Упражнение 4. Генератор простых паролей 31
Заключительные размышления о первой главе 32

Глава 2. Язык данных и выражений **33**
2.1. Типы данных: строительные блоки программ 33
2.2. Переменные как именованные контейнеры 34
2.3. Числа: целые, вещественные и их особенности 35
Целые числа (`int`) 35
Вещественные числа (`float`) 36
Преобразование между типами чисел 36
Математические операции с числами 37
2.4. Строки: работа с текстом в Python 37
Создание и базовые операции со строками 38
Индексация и срезы строк 38
Методы строк для обработки текста 39
Проверочные методы строк 39
Форматирование строк 40

2.5. Булевы значения: логика истины и лжи ... 40
Основы булевых значений ... 40
Истинность различных типов данных ... 41
Логические операторы ... 41
Операторы сравнения ... 42
Особенности работы с булевыми значениями ... 42
2.6. Операторы и приоритет операций ... 43
Арифметические операторы ... 43
Операторы сравнения и их цепочки ... 43
Логические операторы и короткое замыкание ... 44
Операторы присваивания ... 44
2.7. Ввод данных от пользователя ... 45
Базовое использование функции `input()` ... 46
Преобразование типов при вводе ... 46
Создание надежного ввода с проверкой ... 46
Ввод сложных данных ... 47
2.8. Форматирование вывода: красота в деталях ... 48
f-строки: современный стандарт форматирования ... 48
Спецификаторы формата для чисел ... 48
Выравнивание и заполнение ... 49
Форматирование дат и времени ... 49
Создание структурированного вывода ... 50
Многострочное форматирование ... 50
2.9. Упражнения для закрепления ... 51
Упражнение 1. Калькулятор типов данных ... 51
Упражнение 2. Конвертер единиц измерения ... 52
Упражнение 3. Анализатор текста ... 53
Упражнение 4. Система оценки знаний ... 54
Упражнение 5. Личный финансовый помощник ... 55
Рекомендации по выполнению упражнений ... 57
Заключение ... 57

Глава 3. Искусство принятия решений ... 58

3.1. Условные конструкции: развилки в программе ... 58
3.2. Логические операторы: И, ИЛИ, НЕ ... 60
3.3. Сложные условия и их упрощение ... 61
Техника 1. Использование промежуточных переменных ... 61
Техника 2. Использование вложенных условий для упрощения логики ... 62
Техника 3. Принцип раннего возврата ... 62
3.4. Вложенные условия: решения внутри решений ... 63
Техника упрощения: инверсия условий ... 64
Комбинирование вложенных условий с логическими операторами ... 65
3.5. Сопоставление с образцом (`match-case`) ... 65
Сопоставление с несколькими значениями ... 66
Сопоставление с защитными условиями (`guard clauses`) ... 66
Практический пример: калькулятор с `match-case` ... 67
Сопоставление со структурированными данными ... 68
3.6. Практические примеры: калькулятор, игра в угадайку ... 68
Проект 1. Интеллектуальный калькулятор ... 69
Проект 2. Игра «Угадай число» с адаптивным интеллектом ... 71
3.7. Задачи для самостоятельного решения ... 74

Глава 4. Циклы — автоматизация повторений ... 77

4.1. Философия циклов: зачем повторять код ... 77
4.2. Цикл `for`: известное количество итераций ... 77
4.3. Цикл `while`: до тех пор, пока ... 79
4.4. Управление циклами: `break` и `continue` ... 81

Команда `break`: досрочный выход ... 81
Команда `continue`: пропуск итерации ... 83
Практический пример: валидация данных ... 83
4.5. Вложенные циклы: циклы внутри циклов ... 84
Принцип работы вложенных циклов ... 85
Работа с двумерными структурами ... 85
Генерация комбинаций ... 86
Управление выполнением вложенных циклов ... 86
Оптимизация вложенных циклов ... 87
4.6. Генераторы списков: элегантность Python ... 88
Генераторы с условиями ... 88
Сложные выражения в генераторах ... 89
Вложенные генераторы списков ... 89
Условные выражения в генераторах ... 89
Производительность и читаемость ... 90
Ограничения и рекомендации ... 91
4.7. Практические алгоритмы: поиск, сортировка ... 91
Поиск элемента в списке ... 91
Поиск максимального/минимального значения ... 92
Алгоритмы сортировки: пузырьковая сортировка ... 92
Поиск подстроки в строке ... 93
4.8. Большая практическая работа: анализ текста ... 93
Постановка задачи ... 93
Пример реализации ... 93
Объяснение и расширение задачи ... 95

Глава 5. Функции — строительные блоки программ ... 96

5.1. Зачем нужны функции: принцип DRY ... 96
Проблема дублирования кода ... 96
Принцип DRY: Don't Repeat Yourself ... 97
Функции как решение проблемы повторения ... 97
Преимущества использования функций ... 98
Функции в реальном мире программирования ... 98
5.2. Создание собственных функций ... 98
Синтаксис определения функции ... 98
Простейшие примеры функций ... 99
Функции с возвращаемым значением ... 100
Функции с несколькими параметрами ... 100
Практический пример: функция для проверки четности ... 100
Соглашения об именовании функций ... 101
Документирование функций ... 101
5.3. Параметры и аргументы: передача данных ... 102
Параметры vs Аргументы: важное различие ... 102
Позиционные аргументы ... 102
Именованные аргументы ... 103
Смешивание позиционных и именованных аргументов ... 103
Практический пример: функция расчета стоимости доставки ... 104
5.4. Возвращение значений: результат работы функции ... 104
Оператор `return` ... 104
Возврат различных типов данных ... 105
Возврат нескольких значений ... 105
Условный возврат значений ... 106
Досрочный возврат ... 106
Практический пример: функция анализа текста ... 107
5.5. Область видимости переменных ... 108
Локальные переменные ... 108
Глобальные переменные ... 109

Конфликт имен: локальные переменные скрывают глобальные ... 109
Ключевое слово `global` ... 109
Параметры функции — это локальные переменные ... 110
Практический пример: управление банковским счетом ... 110
Рекомендации по работе с областями видимости ... 111
5.6. Значения по умолчанию и именованные параметры ... 112
Параметры со значениями по умолчанию ... 112
Правила использования значений по умолчанию ... 112
Именованные параметры в сочетании со значениями по умолчанию ... 113
Практический пример: функция подключения к базе данных ... 114
5.7. Функции как объекты первого класса ... 115
Присваивание функций переменным ... 115
Передача функций как аргументов ... 116
Функции высшего порядка ... 116
Хранение функций в структурах данных ... 117
Практический пример: система обработки событий ... 118
5.8. Рекурсия: функции, вызывающие сами себя ... 119
Что такое рекурсия ... 120
Простейший пример: факториал ... 120
Рекурсивный обход структур данных ... 120
Числа Фибоначчи ... 121
Рекурсивный поиск файлов ... 122
Важные принципы работы с рекурсией ... 124
Когда использовать рекурсию ... 124
5.9. Практический проект: библиотека математических функций ... 124
Структура проекта ... 124
Рекомендации по реализации ... 125
Рекомендации по дальнейшему развитию ... 126
Глава 6. Структуры данных Python ... 127
6.1. Списки: упорядоченные коллекции данных ... 127
Создание списков ... 127
Индексация и доступ к элементам ... 128
Изменяемость списков ... 128
Особенности работы со ссылками ... 128
Списки в условных выражениях ... 129
Вложенные списки ... 129
6.2. Операции со списками: добавление, удаление, поиск ... 129
Добавление элементов ... 130
Удаление элементов ... 131
Поиск элементов ... 132
6.3. Срезы: элегантное извлечение подпоследовательностей ... 133
Базовый синтаксис срезов ... 133
Работа с отрицательными индексами ... 133
Использование шага в срезах ... 134
Срезы для копирования и модификации ... 134
Срезы строк ... 135
Практические применения срезов ... 135
6.4. Кортежи: неизменяемые последовательности ... 137
Создание кортежей ... 137
Неизменяемость кортежей ... 137
Доступ к элементам кортежей ... 138
Операции с кортежами ... 138
Распаковка кортежей ... 138
Когда использовать кортежи ... 139
Сравнение производительности: список vs кортеж ... 140
Именованные кортежи ... 141

6.5. Словари: ключ-значение в действии ... 141
Создание словарей ... 142
Доступ к значениям ... 142
Изменение словарей ... 143
Методы работы со словарями ... 144
Вложенные словари ... 144
6.6. Множества: уникальность и математические операции ... 145
Создание множеств ... 146
Операции с элементами множества ... 146
Математические операции с множествами ... 147
Практические применения множеств ... 147
Множества и производительность ... 148
Неизменяемые множества (`frozenset`) ... 149
Практические рекомендации по использованию множеств ... 149
6.7. Выбор правильной структуры данных ... 150
Критерии выбора структуры данных ... 150
Практические сценарии выбора ... 151
Практические рекомендации ... 153
6.8. Вложенные структуры: списки словарей и словари списков ... 154
Списки словарей: коллекции записей ... 154
Словари списков: группировка данных ... 155
Рекомендации по дизайну вложенных структур ... 157
6.9. Практический проект: система учета студентов ... 159
Архитектура системы и выбор структур данных ... 159
Управление студентами ... 160
Управление курсами и регистрацией ... 161
Система оценок и академическая успеваемость ... 163
Расписание и управление временем ... 164
Аналитика и отчеты ... 166
Демонстрация системы в действии ... 168
Анализ архитектурных решений ... 170
Расширения и улучшения ... 170

Глава 7. Работа с файлами и данными ... 172

7.1. Файловая система глазами программиста ... 172
Концепция файловой системы ... 172
Пути к файлам: абсолютные и относительные ... 172
Работа с путями в Python ... 173
Операции с файловой системой ... 173
7.2. Чтение и запись текстовых файлов ... 174
Основы работы с файлами: функция `open()` ... 174
Режимы открытия файлов ... 174
Конструкция `with`: безопасная работа с файлами ... 175
Способы чтения файлов ... 175
Запись в файлы ... 175
Обработка кодировок ... 176
Практический пример: анализатор текста ... 176
7.3. Обработка CSV-данных ... 177
Что представляет собой формат CSV ... 178
Модуль csv: стандартные инструменты Python ... 178
Чтение CSV-файлов ... 178
Запись CSV-файлов ... 179
Обработка различных форматов CSV ... 179
7.4. Работа с JSON: современный формат данных ... 180
Что такое JSON и почему он важен ... 180
Модуль json: встроенные возможности Python ... 181
Соответствие типов данных ... 181

Работа с JSON-файлами ... 182
Обработка ошибок при работе с JSON ... 183
7.5. Обработка исключений: что делать, когда что-то идет не так ... 184
Философия обработки ошибок в Python ... 184
Базовая структура `try-except` ... 185
Иерархия исключений в Python ... 185
Обработка множественных исключений ... 186
7.6. Практический проект: анализатор логов веб-сервера ... 187
Понимание задачи ... 188
Архитектура решения ... 188
Парсинг строк лога ... 189
Анализ данных ... 189
Генерация отчетов ... 191
Главная функция и использование ... 193
Файл конфигурации ... 193
Тестирование и расширение ... 194
Глава 8. Модули и пакеты ... 195
8.1. Организация кода: от скрипта к модулю ... 195
Эволюция от монолитного скрипта ... 195
Принципы модульности ... 197
Преимущества модульной организации ... 198
8.2. Импорт модулей: различные способы ... 198
Основы импорта: оператор `import` ... 198
Селективный импорт: `from...import` ... 199
Переименование при импорте: оператор `as` ... 200
Импорт всего содержимого: `from module import *` ... 200
Абсолютные и относительные импорты ... 201
Условный импорт и обработка ошибок ... 202
Отложенный импорт ... 202
Практический пример: система логирования ... 202
Лучшие практики импорта ... 204
8.3. Стандартная библиотека Python: обзор возможностей ... 205
Модуль `os`: взаимодействие с операционной системой ... 205
Модуль `sys`: информация о системе и интерпретаторе ... 206
Модуль `datetime`: работа с датой и временем ... 207
Модули `math` и `random`: математические вычисления ... 208
Модуль `json`: работа с данными JSON ... 209
Интеграция модулей: комплексный пример ... 210
8.4. Создание собственных пакетов ... 211
Анатомия пакета: структура и принципы ... 211
Создание базового пакета ... 212
Создание вложенных пакетов ... 213
Использование созданного пакета ... 215
Относительные импорты внутри пакета ... 215
8.5. Менеджер пакетов `pip`: расширяем возможности ... 216
Философия повторного использования кода ... 216
Основные операции с `pip` ... 216
Управление зависимостями проекта ... 217
Практический пример: работа с внешними библиотеками ... 217
Обновление и удаление пакетов ... 218
Работа с индексами и источниками ... 219
8.6. Виртуальные окружения: изоляция проектов ... 219
Проблема глобального пространства ... 219
venv: стандартный инструмент изоляции ... 219
Альтернативы `venv` ... 220
Лучшие практики работы с виртуальными окружениями ... 221
Автоматизация работы с окружениями ... 221

8.7. Практическая работа: создание утилитарного пакета ... 222
Планирование пакета ... 222
Создание структуры пакета ... 222
Создание модулей пакета ... 223
Использование ... 230
Заключение ... 233

Глава 9. Объектно-ориентированное программирование ... 234

9.1. Философия ООП: моделирование реального мира ... 234
Зачем нужно объектно-ориентированное программирование ... 234
Основные принципы ООП ... 235
Моделирование реального мира ... 235
Преимущества объектно-ориентированного подхода ... 236
9.2. Классы и объекты: шаблоны и экземпляры ... 236
Анатомия класса в Python ... 236
Инициализация объектов: метод `__init__` ... 237
Атрибуты экземпляра ... 237
Методы класса ... 238
Создание более сложного класса ... 239
Важные концепции ... 240
9.3. Атрибуты и методы: состояние и поведение ... 241
Методы экземпляра: поведение объекта ... 241
Загадочный `self`: понимание механизма ... 242
Атрибуты экземпляра vs атрибуты класса ... 242
Изменение атрибутов класса ... 244
Методы как атрибуты объекта ... 244
Динамическое добавление атрибутов и методов ... 245
9.4. Инкапсуляция: сокрытие внутренней реализации ... 245
Проблема прямого доступа к атрибутам ... 245
Приватные атрибуты: соглашения Python ... 246
Геттеры и сеттеры: классический подход ... 248
9.5. Наследование: построение иерархий классов ... 249
Базовые и дочерние классы ... 250
Функция `super()`: связь с родительским классом ... 251
Построение многоуровневой иерархии ... 252
Множественное наследование ... 253
Порядок разрешения методов (MRO) ... 255
9.6. Полиморфизм: одинаковый интерфейс, разное поведение ... 256
Полиморфизм через наследование ... 256
Duck Typing: «Если это ходит как утка...» ... 257
Переопределение методов для специфического поведения ... 259
Практическая польза полиморфизма ... 261
9.7. Специальные методы: магия Python ... 262
Основы создания и представления объектов ... 263
Операторы сравнения и арифметические операции ... 263
Контейнерные методы: работа с коллекциями ... 266
Продвинутые специальные методы ... 267
9.8. Декораторы: модификация поведения ... 269
Встроенные декораторы классов ... 269
Создание пользовательских декораторов ... 272
Декораторы для методов класса ... 274
Композиция декораторов ... 276
9.9. Свойства: контролируемый доступ к атрибутам ... 277
Создание свойств с помощью функции `property()` ... 277
Использование декоратора `@property` ... 278
Свойства с `deleter` ... 280

9.10. Упражнения для закрепления пройденного материала ... 282
Упражнение 1. Моделирование банковского счета ... 282
Упражнение 2. Система управления книгами ... 283
Упражнение 3. Умный термостат ... 283
Упражнение 4. Иерархия транспортных средств ... 283
Упражнение 5. Умная коллекция ... 284

Глава 10. Отладка и тестирование ... 285

10.1. Искусство отладки: стратегии поиска ошибок ... 285
Философия отладки ... 285
Стратегии поиска ошибок ... 285
Чтение и интерпретация трейсбеков ... 287
Типичные ошибки и их интерпретация ... 288
Print-отладка: простой и эффективный метод ... 288
Условная отладка ... 289

10.2. Отладчик Python: пошаговое выполнение ... 290
Введение в pdb ... 290
Основные команды pdb ... 291
Условные точки останова ... 291
Отладка в IDE ... 292
Создание отладочной конфигурации ... 292
Инспекция переменных и состояния программы ... 294
Отладка рекурсивных функций ... 296
Заключение по отладке ... 297

10.3. Логирование: правильный способ отслеживания работы программы ... 297
Философия логирования ... 297
Модуль logging: основы ... 297
Уровни логирования ... 298
Конфигурация логирования ... 299
Фильтры и обработчики ... 301

10.4. Юнит-тестирование: модуль unittest ... 304
Философия тестирования ... 304
Модуль unittest: основы ... 304
Структура тестов ... 305
Методы утверждений (Assertions) ... 308
Пропуск тестов и условное выполнение ... 311
Группировка тестов и тестовые наборы ... 313

10.5. Pytest: современный фреймворк тестирования ... 316
Основы pytest ... 316
Организация тестов ... 317
Фикстуры (Fixtures) ... 318
Параметризация тестов ... 320
Комбинированная параметризация ... 321
Маркеры (Markers) и плагины ... 322
Конфигурация pytest ... 323

10.6. Покрытие кода тестами ... 324
Установка и использование coverage.py ... 324
Типы покрытия ... 326
Интерпретация результатов покрытия ... 327
Улучшение покрытия ... 329
Настройка и конфигурация покрытия ... 331
Интеграция с pytest ... 332
Метрики качества тестирования ... 332
Лучшие практики покрытия ... 333

10.7. Документирование кода: docstring и типы ... 334
Строки документации (Docstrings) ... 334
Типизация в Python ... 334
Статический анализ с MyPy ... 335

Документирование классов ... 336
10.8. Практическая работа: тестирование математической библиотеки ... 337
Планирование математической библиотеки ... 337
Разработка через тестирование (TDD) ... 338
Расширенное тестирование с `Pytest` ... 339
Интеграционные тесты и фикстуры ... 341
Тестирование производительности и граничных случаев ... 341

Глава 11. Работа с внешними данными ... 343
11.1. HTTP-запросы: взаимодействие с веб-сервисами ... 343
Установка и первые шаги ... 343
Анатомия HTTP-запроса ... 343
Работа с различными HTTP-методами ... 344
Обработка ответов сервера ... 344
Обработка ошибок и исключений ... 345
Работа с заголовками и аутентификацией ... 346
Сессии и постоянные соединения ... 346
Практический пример: мониторинг веб-сайта ... 347
11.2. Парсинг HTML: библиотека BeautifulSoup ... 348
Установка и первое знакомство ... 349
Навигация по DOM-дереву ... 349
CSS-селекторы: мощный инструмент поиска ... 350
Извлечение данных и атрибутов ... 352
Обработка текста и очистка данных ... 353
Практический пример: парсинг новостного сайта ... 354
Работа с формами и интерактивными элементами ... 358
Лучшие практики парсинга ... 360
11.3. Работа с API: получение данных из интернета ... 363
REST API и основные принципы ... 363
Работа с JSON-данными ... 364
Аутентификация и API-ключи ... 365
Пагинация: работа с большими наборами данных ... 366
Rate Limiting: ограничения скорости запросов ... 367
Обработка ошибок API ... 368
11.4. Базы данных: SQLite в Python ... 369
Введение в SQLite и модуль sqlite3 ... 369
CRUD-операции: основа работы с данными ... 370
Основы ORM: упрощение работы с базами данных ... 376
11.5. Регулярные выражения: мощный инструмент работы с текстом ... 378
Модуль `re`: основы работы с регулярными выражениями ... 379
Метасимволы: язык шаблонов ... 379
Группы: извлечение и структурирование данных ... 380
Практические примеры: решение реальных задач ... 381

Глава 12. Алгоритмы и структуры данных ... 384
12.1. Сложность алгоритмов: большое O ... 384
Временная сложность: как растет время выполнения ... 384
Нотация большого O ... 385
Пространственная сложность ... 386
Анализ реальных задач ... 388
Амортизированная сложность ... 389
Практические рекомендации ... 390
12.2. Алгоритмы поиска: линейный и бинарный ... 391
Линейный поиск: простота и универсальность ... 391
Бинарный поиск: эффективность через упорядоченность ... 392
Визуализация процесса поиска ... 393
Сравнительный анализ производительности ... 394
Практическое применение и вариации ... 395

Выбор алгоритма поиска ... 397
12.3. Алгоритмы сортировки: от пузырька до быстрой ... 397
Сортировка пузырьком (Bubble Sort) ... 397
Сортировка выбором (Selection Sort) ... 399
Сортировка вставками (Insertion Sort) ... 399
Сортировка слиянием (Merge Sort) ... 400
Быстрая сортировка (Quick Sort) ... 402
Сравнение алгоритмов сортировки ... 403
12.4. Стеки и очереди: LIFO и FIFO ... 404
Стеки: последний пришел — первый ушел ... 404
Очереди: первый пришел — первый ушел ... 406
Эффективная реализация с помощью deque ... 406
Практическое применение: симулятор обслуживания клиентов ... 408
Специализированные реализации в Python ... 410
12.5. Деревья: иерархические данные ... 411
Бинарные деревья: основа древовидных структур ... 411
Алгоритмы обхода деревьев ... 412
Бинарные деревья поиска: эффективная организация данных ... 414
Практическое применение и анализ производительности ... 417
Глава 13. Специализированные области ... 418
13.1. Веб-разработка: Flask как введение ... 418
Установка Flask и первое приложение ... 418
Маршруты: навигация в веб-приложении ... 419
Шаблоны: разделение логики и представления ... 420
Работа с формами ... 422
REST API: создание веб-сервисов ... 425
13.2. Анализ данных: pandas и numpy ... 428
Установка и первое знакомство ... 428
Загрузка данных из файлов ... 429
Исследование структуры данных ... 429
Индексация и выборка данных ... 430
Группировка и агрегация данных ... 430
Обработка пропущенных значений ... 431
Создание новых столбцов и трансформации ... 431
Базовые статистические операции ... 432
Простая визуализация данных ... 432
Практический пример: анализ продаж ... 433
Временные ряды и даты ... 433
Резюме и лучшие практики ... 434
13.3. Машинное обучение: scikit-learn ... 434
Подготовка рабочей среды ... 434
Классификация: определение категорий ... 435
Регрессия: предсказание непрерывных значений ... 436
Кластеризация: поиск скрытых групп ... 437
Оценка и улучшение моделей ... 438
13.4. Выбор специализации: куда двигаться дальше ... 439
Карта возможностей: основные направления специализации ... 439
Методика самооценки: определяем свои склонности ... 440
Построение траектории обучения ... 441
Развитие soft skills и профессиональных навыков ... 443
Заключение ... 445
Путь, который мы прошли вместе ... 445
Python как первый, но не последний язык ... 445
Влияние программирования на мышление ... 446
Ответственность разработчика ... 446
Заключительные размышления ... 446

ПРЕДИСЛОВИЕ

Добро пожаловать в мир программирования

Вы держите в руках не просто учебник по языку программирования Python — это приглашение в удивительный мир создания программного обеспечения, где абстрактные идеи превращаются в работающие решения реальных проблем. Программирование сегодня стало одним из самых мощных инструментов человеческого интеллекта, позволяющим автоматизировать процессы, анализировать огромные объемы данных, создавать интерактивные приложения и даже моделировать сложные системы.

Эта книга создана с глубоким убеждением, что программирование должно преподаваться не как набор технических приемов или заучивание синтаксиса, а как дисциплина мышления, формирующая способность к систематическому решению проблем. Каждый программист, независимо от специализации, должен понимать фундаментальные принципы вычислений, алгоритмические подходы к решению задач и культуру создания качественного кода.

ФИЛОСОФИЯ ИЗУЧЕНИЯ ПРОГРАММИРОВАНИЯ

Программирование как способ мышления

Многие люди воспринимают программирование исключительно как техническую дисциплину — изучение синтаксиса языка, запоминание функций и методов. Такой подход приводит к формированию поверхностных знаний, которые быстро устаревают с появлением новых технологий. В действительности программирование представляет собой фундаментальный способ мышления, который можно применить к решению задач в любой области человеческой деятельности.

Программистское мышление характеризуется несколькими ключевыми особенностями. Во-первых, это системный подход к декомпозиции сложных проблем — умение разбивать сложную задачу на множество более простых подзадач, каждую из которых можно решить независимо. Во-вторых, это алгоритмическое мышление — способность представить процесс решения в виде последовательности четко определенных шагов. В-третьих, это абстрактное мышление — умение выделять общие закономерности и создавать универсальные решения, применимые к широкому классу задач.

Понимание вместо запоминания

Традиционные методы обучения программированию часто строятся по принципу «покажи и повтори»: студенту демонстрируют пример кода, а затем просят написать нечто похожее. Такой подход может дать быстрые результаты на начальном этапе, но создает ложное ощущение понимания и не формирует устойчивых навыков.

В этой книге мы следуем принципиально иному подходу. Каждая концепция объясняется с трех позиций: *что* представляет собой данная конструкция, *почему* она работает именно таким образом и *когда* ее следует применять. Такая трехмерная модель понимания позволяет не просто запомнить синтаксис, но и развить интуицию относительно эффективных способов решения различных классов задач.

Например, изучая циклы, мы не ограничиваемся демонстрацией синтаксиса `for` и `while`. Мы исследуем концептуальную природу итерации, анализируем различные паттерны (patterns) использования циклов, обсуждаем производительность различных подходов и рассматриваем альтернативные способы достижения того же результата. Такой подход формирует глубокое понимание, которое остается актуальным независимо от изменений в технологиях.

Культура качественного кода

Одним из наиболее важных аспектов профессионального программирования является культура создания качественного кода. К сожалению, многие учебные материалы игнорируют этот аспект, сосредотачиваясь исключительно на функциональности программ. В результате у начинающих программистов часто формируются вредные привычки, от которых потом крайне сложно избавиться.

Качественный код характеризуется не только корректностью работы, но и читаемостью, поддерживаемостью, эффективностью и элегантностью решения. Читаемость подразумевает, что код должен быть понятен другим программистам (и вам самим через несколько месяцев). Поддерживаемость означает возможность легкого внесения изменений и расширения функциональности. Эффективность касается разумного использования вычислительных ресурсов. Элегантность отражает красоту и простоту решения — качество, которое приходит с опытом и глубоким пониманием предметной области.

В этой книге вопросы качества кода не выносятся в отдельную главу, а интегрированы в каждый раздел. С самых первых примеров демонстрируются правильные практики именования переменных (variables), структурирования кода, написания комментариев (comments) и документации (documentation). Такой подход позволяет сформировать правильные привычки с самого начала обучения.

Почему Python — идеальный первый язык

Философия простоты и читаемости

Создатели Python руководствовались философией «код должен быть написан в том числе для чтения людьми, а не только для выполнения компьютерами». Эта философия кардинально отличает Python от многих других языков программирования, где синтаксические конструкции могут быть крайне сложными для понимания. Принцип читаемости кода в Python не является случайным — он отражает глубокое понимание того, что программирование в первую очередь является интеллектуальной деятельностью, требующей ясности мышления.

Синтаксис Python максимально приближен к естественному английскому языку. Приведем пример решения одной из задач в Python:

```
if age >= 21:
    print("Вы можете голосовать")
else:
    print("Вы пока не можете голосовать")
```

Даже человек, никогда не изучавший программирование, может интуитивно понять смысл этого кода. Такая естественность синтаксиса позволяет сосредоточиться на изучении алгоритмического мышления, а не на запоминании сложных синтаксических правил.

Мультипарадигмальность как преимущество

Python поддерживает несколько парадигм программирования: процедурную, объектно-ориентированную и функциональную. Для начинающего программиста это представляет огромную ценность, поскольку позволяет естественным образом переходить от простых линейных программ к более сложным архитектурным решениям.

На начальном этапе обучения студенты работают в процедурной парадигме, создавая простые последовательности команд. По мере накопления опыта они начинают группировать код в функции (functions), что соответствует функциональной парадигме. Затем, при решении более сложных задач, естественным образом приходит понимание необходимости объектно-ориентированного программирования (object-oriented programming). Такая эволюция подходов происходит органично, без резких переходов и концептуальных скачков.

Обширная экосистема и практическое применение

Python обладает одной из самых богатых экосистем среди всех языков программирования. Стандартная библиотека Python содержит модули для решения широчайшего спектра задач: от работы с файлами и сетевыми протоколами до математических вычислений и обработки данных. Дополнительные библиотеки, доступные через систему управления пакетами (package management system), расширяют возможности языка практически до бесконечности.

Эта особенность Python имеет огромное педагогическое значение. Студенты с самого начала обучения могут работать с реальными данными, создавать веб-приложения (web applications), анализировать статистику, обрабатывать изображения или даже создавать простые игры. Возможность быстро получать впечатляющие результаты поддерживает мотивацию к обучению и демонстрирует практическую ценность приобретаемых знаний.

Плавная кривая обучения

Python спроектирован таким образом, что базовые концепции можно изучать постепенно, без необходимости сразу погружаться в сложные технические детали. Например, при изучении функций студенту не нужно сразу понимать особенности управления памятью или сложности системы типов. Эти продвинутые темы можно изучать по мере необходимости, когда базовые навыки уже сформированы.

Одновременно Python не скрывает сложность искусственно. По мере развития навыков программирования студент может изучать все более продвинутые

возможности языка: декораторы (decorators), генераторы (generators), метаклассы (metaclasses) и другие мощные конструкции. Такая архитектура языка позволяет использовать его как для обучения основам программирования, так и для создания сложных промышленных систем.

Как правильно использовать книгу

Структура изложения материала

Книга построена по принципу концентрических кругов — каждая новая глава расширяет и углубляет знания, полученные в предыдущих главах. Такой подход позволяет избежать информационной перегрузки и обеспечивает постепенное формирование целостной картины программирования на Python.

Каждая глава состоит из трех основных компонентов. **Теоретическая часть** содержит объяснение концепций, принципов и теоретических основ изучаемой темы. Здесь рассматриваются не только технические аспекты, но и контекст применения, исторические предпосылки появления тех или иных конструкций языка, а также связи с общими принципами информатики. **Демонстрационные примеры** представляют собой тщательно подобранные фрагменты кода, иллюстрирующие применение изучаемых концепций. Каждый пример сопровождается подробными комментариями, объясняющими логику работы программы, выбор конкретных решений и возможные альтернативы. **Практические упражнения** предлагают задачи различного уровня сложности — от простых упражнений на закрепление синтаксиса до творческих заданий, требующих интеграции знаний из разных областей. Особое внимание уделяется задачам, имеющим практическое значение и демонстрирующим реальное применение программирования.

Рекомендации по изучению

Эффективное изучение программирования требует активного участия и регулярной практики. Простое чтение кода и примеров не может заменить собственного опыта написания программ. Каждый пример, представленный в книге, должен быть не только изучен, но и выполнен на компьютере. Более того, рекомендуется экспериментировать с примерами — изменять параметры, добавлять новую функциональность, исследовать граничные случаи.

Особое внимание следует уделить выполнению практических упражнений. Начинайте с простых задач и постепенно переходите к более сложным. Если задача кажется слишком трудной, не стесняйтесь вернуться к предыдущим главам и повторить соответствующий материал. Программирование — кумулятивная дисциплина, где каждая новая концепция строится на основе предыдущих знаний.

Не стремитесь запоминать синтаксис наизусть. Вместо этого сосредоточьтесь на понимании принципов и закономерностей. Хороший программист знает не все функции стандартной библиотеки, но умеет быстро находить нужную информацию в документации и применять ее для решения конкретных задач. **Не бойтесь гуглить и искать ответы самостоятельно** — это неотъемлемая часть жизни любого программиста. Даже опытные разработчики регулярно обращаются к поисковикам, документации, форумам и Stack Overflow. Уме-

ние формулировать правильные вопросы и находить релевантную информацию — это такой же важный навык, как и написание кода. *Самостоятельное исследование проблемы часто приводит к более глубокому пониманию материала*, чем получение готового ответа. **Развивайте навыки исследования и анализа.** В данной книге вы встретите примеры различной степени сложности. Некоторые из них намеренно включают техники и концепции, которые еще не были подробно объяснены. Это сделано для того, чтобы стимулировать ваше любопытство и желание разобраться в неизвестном коде. Когда вы сталкиваетесь с непонятным фрагментом, попробуйте:

- выполнить код по частям и посмотреть на промежуточные результаты;
- поискать информацию о незнакомых функциях или методах;
- поэкспериментировать с параметрами и посмотреть, как изменяется поведение;
- найти аналогичные примеры в интернете.

Помните: столкнуться с непонятным кодом и потратить время на его изучение — это нормально. Именно так происходит реальная работа программиста. Способность разбираться в чужом коде, изучать новые библиотеки и адаптироваться к незнакомым техникам — ключевые профессиональные навыки, которые развиваются только через практику и постоянное обучение.

Работа с ошибками как инструмент обучения

Ошибки являются неотъемлемой частью процесса программирования, и начинающие разработчики должны научиться воспринимать их не как неудачи, а как возможности для обучения. Python предоставляет чрезвычайно информативные сообщения об ошибках, которые помогают локализовать и исправить проблемы в коде.

В этой книге мы уделяем особое внимание анализу типичных ошибок. Многие примеры специально содержат потенциальные проблемы, которые обсуждаются и исправляются. Такой подход помогает развить навыки отладки (debugging) — одного из важнейших умений профессионального программиста.

Рекомендуется вести журнал встретившихся ошибок с описанием их причин и способов исправления. Со временем вы заметите, что многие ошибки повторяются, и научитесь предотвращать их на этапе написания кода.

Использование дополнительных ресурсов

Хотя эта книга содержит достаточно подробную информацию, необходимую для освоения основ программирования на Python, изучение любого языка программирования не должно ограничиваться одним источником. Официальная документация Python (python.org) является исчерпывающим справочником по всем возможностям языка. Научитесь пользоваться документацией — это навык, который понадобится на протяжении всей карьеры программиста.

Интерактивная оболочка Python (Python REPL — Read-Eval-Print Loop) представляет собой мощный инструмент для экспериментирования и изучения языка. Используйте ее для проверки гипотез, тестирования фрагментов кода и исследования поведения различных функций и методов.

Онлайн-сообщества программистов, такие как Stack Overflow, предоставляют доступ к коллективному опыту миллионов разработчиков. Научитесь формулировать вопросы правильно и искать ответы на уже заданные вопросы — эти навыки существенно ускорят ваше обучение.

Настройка среды разработки

Выбор интерпретатора Python

Прежде чем приступить к изучению программирования, необходимо настроить рабочую среду. Python является интерпретируемым языком, что означает необходимость установки интерпретатора (interpreter) — программы, которая выполняет код Python. Рекомендуется использовать официальную версию Python, доступную на сайте python.org, а не альтернативные реализации или дистрибутивы.
На момент написания этой книги актуальной является версия Python 3.12, однако все примеры совместимы с Python 3.8 и более поздними версиями. Важно убедиться, что вы устанавливаете Python 3.x, а не Python 2.x, поскольку Python 2 официально прекратил поддерживаться и содержит существенные отличия в синтаксисе.

Установка Python в различных операционных системах

Windows. Загрузите установщик с официального сайта python.org. При установке обязательно отметьте опцию Add Python to PATH — это позволит запускать Python из командной строки любой папки. Рекомендуется также установить опцию Install for all users, если у вас есть соответствующие права доступа.
macOS. На современных версиях macOS Python 3 может быть предустановлен, однако рекомендуется установить актуальную версию с официального сайта. Альтернативно можно использовать менеджер пакетов Homebrew:

```
brew install python
```

Linux. В большинстве дистрибутивов Linux Python 3 предустановлен. Для установки дополнительных компонентов используйте стандартный менеджер пакетов вашего дистрибутива:

```
sudo apt install python3 python3-pip python3-venv
# CentOS/RHEL/Fedora
sudo yum install python3 python3-pip python3-venv
```

Выбор среды разработки

Для эффективной работы с Python необходима качественная среда разработки. Существует множество вариантов, от простых текстовых редакторов до полноценных интегрированных сред разработки (IDE — Integrated Development Environment). Выбор зависит от личных предпочтений и уровня подготовки.
IDLE — это простая среда разработки, поставляемая вместе с Python. Она идеально подходит для начинающих, поскольку не требует дополнительной настройки и содержит все необходимые инструменты: редактор кода с подсвет-

кой синтаксиса, интерактивную оболочку и отладчик. IDLE автоматически проверяет синтаксис кода и предоставляет базовые функции автодополнения.
Visual Studio Code — это бесплатный редактор кода от Microsoft, который благодаря расширениям превращается в мощную среду разработки Python. Основные преимущества включают интеллектуальное автодополнение, интегрированный терминал, систему контроля версий и поддержку отладки. Для работы с Python необходимо установить расширение Python Extension.
PyCharm — профессиональная IDE от компании JetBrains, специально предназначенная для разработки на Python. Существуют как бесплатная Community Edition, так и платная Professional Edition. PyCharm предоставляет расширенные возможности рефакторинга кода, профилирования производительности, работы с базами данных и веб-разработки.
Для изучения основ программирования рекомендуется начать с IDLE или Visual Studio Code. Эти среды предоставляют необходимую функциональность, не перегружая интерфейс продвинутыми возможностями, которые могут отвлекать от изучения языка.

Настройка системы управления пакетами

Python использует систему управления пакетами pip (pip installs packages) для установки дополнительных библиотек. Pip обычно устанавливается автоматически вместе с Python, но важно убедиться в его корректной работе.
Проверить установку pip можно следующей командой:

```
pip –version
```

Для установки пакетов используется команда:

```
pip install package_name
```

Рекомендуется сразу обновить `pip` до актуальной версии:

```
pip install --upgrade pip
```

Создание виртуального окружения

Виртуальные окружения (virtual environments) являются важнейшим инструментом профессиональной разработки на Python. Они позволяют создавать изолированные среды для различных проектов, избегая конфликтов между версиями библиотек.
Для создания виртуального окружения используйте модуль `venv`:

```
python -m venv learning_python
```

Активация виртуального окружения различается в зависимости от операционной системы:

```
# Windows
learning_python\Scripts\activate
# macOS/Linux
source learning_python/bin/activate
```

После активации командная строка должна показывать название окружения в скобках. Все пакеты, установленные в активном виртуальном окружении, будут изолированы от системной установки Python.

Проверка установки

После завершения настройки среды разработки выполните простую проверку работоспособности системы. Создайте файл с именем hello.py и добавьте следующий код:

```
# Простая программа для проверки установки Python
print("Hello, Python!")
print("Версия Python:", end=" ")
import sys
print(sys.version)
# Проверка основных математических операций
result = 2 + 2
print(f"2 + 2 = {result}")
# Проверка работы со списками
numbers = [1, 2, 3, 4, 5]
print(f"Список чисел: {numbers}")
print(f"Сумма чисел: {sum(numbers)}")
```

Запустите программу командой:

```
python hello.py
```

Если программа выполнилась без ошибок и вывела ожидаемые результаты, среда разработки настроена корректно, и вы готовы к изучению программирования на Python.

Добро пожаловать в увлекательное путешествие в мир программирования! Помните, что изучение программирования — это марафон, а не спринт. Будьте терпеливы, не бойтесь экспериментировать и помните, что каждая решенная задача приближает вас к мастерству. Программирование — это не только техническая дисциплина, но и искусство создания элегантных решений сложных проблем. Наслаждайтесь процессом обучения и открывайте для себя безграничные возможности, которые дает владение языком Python.

Глава 1. Первые шаги в мире программирования

1.1. Что такое программирование и зачем оно нужно

Программирование — это искусство и наука создания инструкций для компьютера. Но подобное определение, хотя и технически корректное, не раскрывает истинной сути этой деятельности. Программирование в современном мире представляет собой универсальный инструмент решения проблем, способ формализации мышления и мощный механизм автоматизации интеллектуальной деятельности.

Когда мы программируем, мы на самом деле переводим наши идеи и алгоритмы решения задач на язык, понятный компьютеру. Этот процесс требует от нас предельной точности в формулировках, логической последовательности в рассуждениях и способности разбивать сложные проблемы на более простые составляющие. Именно поэтому изучение программирования развивает аналитическое мышление даже у тех, кто не планирует стать профессиональным разработчиком.

В современном мире программирование перестало быть уделом исключительно специалистов по информационным технологиям. Биологи используют код для анализа ДНК-последовательностей, экономисты — для моделирования финансовых рынков, социологи — для обработки данных опросов, а художники — для создания цифрового искусства. Программирование стало метанавыком, который усиливает эффективность практически в любой области человеческой деятельности.

Важно понимать, что программирование — это не заучивание синтаксиса (`syntax`) конкретного языка. Синтаксис можно выучить за несколько недель, но настоящее мастерство программирования заключается в развитии способности мыслить алгоритмически: видеть закономерности, выявлять повторяющиеся паттерны, разбивать большие задачи на управляемые фрагменты и создавать эффективные решения.

Программирование учит нас быть точными в формулировках. Компьютер не умеет угадывать наши намерения — он выполняет именно то, что мы написали, а не то, что мы имели в виду. Эта особенность заставляет программистов думать предельно четко и учитывать все возможные сценарии развития событий. Такой навык оказывается бесценным не только в работе с кодом, но и в повседневной жизни.

1.2. Python — философия простоты и элегантности

Python занимает особое место среди языков программирования благодаря своей философии. Основные принципы этой философии — простота, читаемость и элегантность — делают Python идеальным языком для изучения основ программирования.

Создатель Python Гвидо ван Россум (Guido van Rossum) сформулировал ключевой принцип языка: «Читаемость кода имеет значение» (Readability counts). В отличие от многих других языков программирования, которые позволяют писать очень сжатый, но трудночитаемый код, Python поощряет создание программ, которые легко понимать не только их автору, но и другим разработчикам.

Синтаксис Python максимально приближен к естественному английскому языку. Вместо фигурных скобок и точек с запятой, которые используются во многих других языках, Python использует отступы (indentation) для структурирования кода. Это не просто стилистическое решение — такой подход принуждает программистов писать хорошо структурированный код с самого начала обучения.

Рассмотрим простой пример, демонстрирующий элегантность Python:

```
# Программа для подсчета количества слов в тексте
text = "Python делает программирование понятным и приятным"
words = text.split()
word_count = len(words)
print(f"В тексте {word_count} слов")
```

Даже человек, никогда не программировавший, может понять, что делает этот код. Такая ясность выражения — одно из главных преимуществ Python.

Python является интерпретируемым языком (interpreted language), что означает интерактивность процесса разработки. Вы можете экспериментировать с кодом, тестировать отдельные функции и немедленно видеть результат — это значительно ускоряет процесс обучения и разработки.

Важным аспектом философии Python является принцип «батарейки включены» (batteries included). Стандартная библиотека Python содержит огромное количество готовых модулей для решения типичных задач: работа с файлами, сетевые протоколы, обработка данных, создание веб-приложений и многое другое. Это позволяет сосредоточиться на решении конкретной задачи, не тратя время на реализацию базовой функциональности.

Python также славится своим сообществом. Экосистема языка включает множество высококачественных внешних библиотек для специализированных задач: NumPy для научных вычислений, Django для веб-разработки, TensorFlow для машинного обучения. Это означает, что по мере роста вашего мастерства вы сможете легко находить готовые решения для сложных задач.

1.3. Интерпретатор как ваш первый собеседник

Интерпретатор (interpreter) Python — это программа, которая читает и выполняет ваш код построчно. В отличие от компилируемых языков (compiled languages), где код сначала переводится в машинный код, а затем выполняется, Python позволяет вам взаимодействовать с программой в режиме реального времени.

Когда вы запускаете интерпретатор Python, вы видите приглашение командной строки, обозначаемое тремя символами «больше»: >>>. Это ваше окно в мир программирования, ваш первый диалог с компьютером на языке, который понимают оба собеседника.

Интерактивный режим (interactive mode) или REPL (Read-Eval-Print Loop — цикл «чтение-вычисление-вывод») позволяет вам экспериментировать с кодом без необходимости создавать отдельные файлы. Вы вводите выражение (expression), интерпретатор его вычисляет и немедленно показывает результат:

```
>>> 2 + 3
5
```

```
>>> name = "Анна"
>>> print(f"Привет, {name}!")
Привет, Анна!
>>> len("Программирование")
16
```

Такой подход особенно ценен на этапе обучения, поскольку позволяет немедленно проверить понимание каждой концепции. Вы можете задать вопрос компьютеру и сразу получить ответ — это создает ощущение живого диалога с машиной.

Интерпретатор также служит отличным калькулятором. Вместо того чтобы доставать телефон для простых вычислений, многие программисты просто открывают Python:

```
>>> 15 * 24 + 78
438
>>> import math
>>> math.sqrt(144)
12.0
>>> 2 ** 10  # 2 в степени 10
1024
```

Важная особенность интерпретатора — его память о предыдущих действиях в рамках одной сессии. Переменные (variables), которые вы определили, остаются доступными:

```
>>> price = 1250
>>> discount = 0.15
>>> final_price = price * (1 - discount)
>>> print(f"Цена со скидкой: {final_price} рублей")
Цена со скидкой: 1062.5 рублей
```

Интерпретатор Python также предоставляет встроенную систему помощи. Функция `help()` дает подробную информацию о любом объекте в Python:

```
>>> help(len)  # получить справку о функции len
>>> help(str)  # узнать о методах строк
```

Для быстрой справки о методах объекта можно использовать функцию dir():

```
>>> dir("hello")  # показать все методы строки
```

Эти инструменты превращают интерпретатор в мощную среду для изучения и экспериментирования. Вместо того чтобы запоминать всё наизусть, вы учитесь задавать правильные вопросы и находить ответы в самой системе.

1.4. Анатомия ошибок: учимся понимать компьютер

Ошибки (errors) в программировании неизбежны — они являются естественной частью процесса разработки. Более того, умение читать и понимать сообщения об ошибках представляет собой один из важнейших навыков программиста. Python отличается особенно информативными сообщениями об ошибках, которые помогают не только найти проблему, но и понять ее причину.

Важно изменить свое отношение к ошибкам. Они не являются свидетельством вашей неспособности — это форма обратной связи от интерпретатора, способ компьютера объяснить, почему он не может выполнить ваши инструкции. Каждое сообщение об ошибке содержит ценную информацию для диагностики и исправления проблемы.

Python классифицирует ошибки по типам. Рассмотрим наиболее распространенные.

- **Синтаксические ошибки** (`SyntaxError`) возникают, когда интерпретатор не может разобрать (`parse`) ваш код:

```
>>> print("Привет, мир!"
  File "<stdin>", line 1
    print("Привет, мир!"
                        ^
SyntaxError: '(' was never closed
```

Обратите внимание на полезность этого сообщения: интерпретатор не только указывает на строку с ошибкой, но и пытается объяснить проблему (`"скобка не была закрыта"`) и даже показывает стрелочкой место, где ожидал найти решение.

- **Ошибки имен** (`NameError`) происходят при обращении к несуществующей переменной:

```
>>> print(message)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'message' is not defined
```

Это сообщение дает нам информацию о том, что переменная `message` не была определена до ее использования.

- **Ошибки типов** (`TypeError`) возникают при попытке выполнить операцию с объектами несовместимых типов:

```
>>> "hello" + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Python ясно объясняет: нельзя складывать строку с числом.

- **Ошибки значений** (`ValueError`) происходят, когда функция получает аргумент правильного типа, но с неподходящим значением:

```
>>> int("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
```

Структура сообщения об ошибке в Python следует определенному формату. **Трассировка** (`traceback`) показывает путь выполнения программы до момента возникновения ошибки. В интерактивном режиме она обычно короткая, но в больших программах может показывать цепочку вызовов функций. Затем указываются **тип ошибки** и **описывающее сообщение**, объясняющее, что именно пошло не так.

Эффективная стратегия работы с ошибками включает несколько шагов.

1. **Не паникуйте.** Ошибки — это нормальная часть процесса программирования.
2. **Внимательно читайте сообщение.** Python старается быть максимально информативным.
3. **Обращайте внимание на номер строки.** Интерпретатор указывает, где искать проблему.
4. **Анализируйте тип ошибки.** Разные типы ошибок требуют разных подходов к исправлению.
5. **Проверьте очевидные проблемы:** закрытые скобки, правильность имен переменных, совместимость типов данных.

Полезная практика — намеренное создание ошибок для изучения их сообщений:

```
>>> # Попробуйте эти примеры и изучите сообщения об ошибках
>>> print(undefined_variable)
>>> 10 / 0
>>> [1, 2, 3][5]
>>> {"a": 1}["b"]
```

Понимание ошибок превращает их из препятствий в союзников. Они становятся вашими учителями, указывающими путь к правильному решению.

1.5. Первая программа: традиция "Hello, World!" по-русски

Традиция написания программы `"Hello, World!"` уходит корнями в 1970-е годы, когда Брайан Керниган впервые использовал этот пример в документации к языку C. С тех пор каждый программист начинает свое знакомство с новым языком именно с этой простой, но символически важной программы. Она служит своеобразным мостом между человеком и машиной — первым успешным актом коммуникации на новом языке.

В Python создание программы `"Hello, World!"` демонстрирует философию простоты языка. То, что в других языках программирования может потребовать нескольких строк кода с импортом библиотек и объявлением функций, в Python выражается одной строкой:

```
print("Hello, World!")
```

Но давайте адаптируем эту классическую программу под русский язык и культурный контекст:

```
print("Привет, мир!")
```

Эта простота обманчива. За кажущейся элементарностью скрывается сложная работа интерпретатора Python. Когда вы вызываете функцию `print()`, происходит несколько важных процессов: Python определяет тип переданного аргумента (в данном случае — строка), преобразует его во внутреннее представление, отправляет данные в стандартный поток вывода (`stdout`), и операционная система отображает результат на экране.

Расширим нашу первую программу, добавив элементы интерактивности:

```
# Наша первая интерактивная программа
name = input("Как вас зовут? ")
print(f"Привет, {name}! Добро пожаловать в мир Python!")

# Добавим немного арифметики
age = int(input("Сколько вам лет? "))
print(f"Через 10 лет вам будет {age + 10} лет.")
```

Этот пример демонстрирует несколько фундаментальных концепций программирования: ввод данных от пользователя, сохранение данных в переменных, преобразование типов данных и форматированный вывод. Функция `input()` позволяет программе взаимодействовать с пользователем, создавая основу для более сложных интерактивных приложений.

Важно отметить использование f-строк (`f-strings`) — современного способа форматирования строк в Python. Префикс f перед строкой позволяет вставлять значения переменных прямо в текст с помощью фигурных скобок. Это делает код более читаемым и эффективным по сравнению со старыми методами конкатенации (`concatenation`) строк.

Функция `int()` демонстрирует концепцию преобразования типов данных. Функция `input()` всегда возвращает строку, даже если пользователь ввел число. Чтобы выполнить математические операции, нам нужно явно преобразовать строку в число.

Создадим более содержательную версию нашей первой программы:

```
# Программа знакомства с элементами вычислений
print("=== Добро пожаловать в Python! ===")
print()

name = input("Представьтесь, пожалуйста: ")
print(f"Приятно познакомиться, {name}!")

# Простая математическая задачка
print("\nДавайте решим простую задачу.")
a = float(input("Введите первое число: "))
b = float(input("Введите второе число: "))

result = a + b
print(f"\n{a} + {b} = {result}")

# Демонстрация других операций
print(f"{a} * {b} = {a * b}")
print(f"{a} / {b} = {a / b:.2f}")  # округление до двух знаков

print(f"\nСпасибо за использование программы, {name}!")
```

На протяжении данного сегмента было введено понятие комментариев (`comments`) — строк, начинающихся с символа #. Комментарии игнорируются интерпретатором и служат для объяснения кода человеку. Хорошие комментарии — это инвестиция в будущее, они помогают вам и другим программистам понять логику программы спустя какое-то время.

1.6. Интерактивный режим vs написание скриптов

Понимание различий между интерактивным режимом и выполнением скриптов (`scripts`) критически важно для эффективной работы с Python. Каждый подход имеет свои преимущества и область применения, и опытный программист знает, когда использовать каждый из них.

Интерактивный режим идеально подходит для экспериментирования, быстрой проверки идей и изучения новых функций языка. Когда вы запускаете интерпретатор Python и видите приглашение `>>>`, вы находитесь в REPL-цикле. Каждое введенное выражение немедленно вычисляется, и результат отображается на экране:

```
>>> x = 10
>>> y = 20
>>> x + y
30
>>> type(x)
<class 'int'>
>>> help(print)
```

Интерактивный режим сохраняет состояние между командами — переменные, определенные в одной строке, доступны в следующих. Это позволяет строить сложные вычисления пошагово, проверяя промежуточные результаты:

```
>>> data = [1, 2, 3, 4, 5]
>>> sum(data)
15
>>> average = sum(data) / len(data)
>>> average
3.0
>>> f"Среднее значение: {average}"
'Среднее значение: 3.0'
```

Скрипты представляют собой файлы с расширением `.py`, содержащие последовательность команд Python. Скрипты выполняются целиком, от первой строки до последней, без возможности промежуточного взаимодействия (если только это не предусмотрено программой).

Создадим файл `calculator.py`:

```
# calculator.py - простой калькулятор
print("Калькулятор Python")
print("-" * 20)

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

# Основная программа
num1 = float(input("Первое число: "))
num2 = float(input("Второе число: "))
```

```
print(f"Сумма: {add(num1, num2)}")
print(f"Произведение: {multiply(num1, num2)}")
```

Для выполнения скрипта используется команда в терминале:

```
python calculator.py
```

Ключевые различия между подходами:

- **Время жизни данных:** в интерактивном режиме переменные существуют до закрытия сессии. В скрипте все переменные удаляются после завершения программы.
- **Отладка (debugging):** интерактивный режим позволяет проверить значение любой переменной в любой момент. В скрипте нужно явно добавлять команды вывода для отладки.
- **Повторное использование:** скрипты можно запускать многократно с разными данными. Интерактивный режим требует повторного ввода всех команд.
- **Сложность:** интерактивный режим удобен для простых операций. Сложные программы лучше писать как скрипты.

Эффективная стратегия разработки часто включает комбинацию обоих подходов:

1. **Экспериментирование в интерактивном режиме:** тестирование новых идей, изучение библиотек, проверка синтаксиса.
2. **Прототипирование в интерактивном режиме:** разработка отдельных функций и алгоритмов.
3. **Перенос в скрипт:** когда логика отработана, код переносится в файл для постоянного использования.
4. **Отладка через интерактивный режим:** возврат к REPL для исследования ошибок.

Рассмотрим практический пример такого `workflow`:

```
# Эксперимент в интерактивном режиме
>>> text = "Python — отличный язык программирования"
>>> words = text.split()
>>> len(words)
5
>>> [len(word) for word in words]
[6, 1, 8, 4, 14]
```

После успешного эксперимента создаем скрипт `word_analyzer.py`:

```
# word_analyzer.py - анализатор текста
def analyze_text(text):
    """Анализирует текст и возвращает статистику"""
    words = text.split()
    word_count = len(words)
    word_lengths = [len(word) for word in words]
    avg_length = sum(word_lengths) / word_count

    return {
        'word_count': word_count,
```

```
        'avg_length': round(avg_length, 2),
        'longest_word': max(words, key=len)
    }

# Основная программа
if __name__ == "__main__":
    user_text = input("Введите текст для анализа: ")
    stats = analyze_text(user_text)

    print(f"Количество слов: {stats['word_count']}")
    print(f"Средняя длина слова: {stats['avg_length']}")
    print(f"Самое длинное слово: {stats['longest_word']}")
```

1.7. Практические упражнения и размышления

Теоретические знания становятся настоящими навыками только через практику. В этом разделе мы предложим серию упражнений, которые помогут закрепить изученные концепции и развить алгоритмическое мышление. Каждое упражнение сопровождается размышлениями о том, какие именно навыки оно развивает.

Упражнение 1. Персонализированное приветствие

Создайте программу, которая запрашивает у пользователя имя, возраст и любимый цвет, а затем создает персонализированное сообщение с использованием всех введенных данных.

Пример решения:

```
# Персонализированное приветствие
name = input("Как вас зовут? ")
age = int(input("Сколько вам лет? "))
color = input("Какой ваш любимый цвет? ")

print(f"Привет, {name}!")
print(f"Мне нравится, что в {age} лет вы изучаете программирование.")
print(f"И кстати, {color} – прекрасный выбор цвета!")

# Бонус: простое вычисление
birth_year = 2025 - age
print(f"Значит, вы родились примерно в {birth_year} году.")
```

Размышления: это упражнение развивает навыки работы с вводом данных, преобразованием типов и форматированием строк. Важно обратить внимание на необходимость преобразования возраста из строки в число для математических операций.

Упражнение 2. Простой калькулятор с проверкой

Напишите программу-калькулятор, которая может выполнять четыре основные арифметические операции и корректно обрабатывает деление на ноль.

```
# Безопасный калькулятор
print("=== Калькулятор Python ===")
```

```
try:
    a = float(input("Введите первое число: "))
    operation = input("Введите операцию (+, -, *, /): ")
    b = float(input("Введите второе число: "))

    if operation == "+":
        result = a + b
    elif operation == "-":
        result = a - b
    elif operation == "*":
        result = a * b
    elif operation == "/":
        if b != 0:
            result = a / b
        else:
            print("Ошибка: деление на ноль невозможно!")
            result = None
    else:
        print("Неизвестная операция!")
        result = None

    if result is not None:
        print(f"Результат: {a} {operation} {b} = {result}")

except ValueError:
    print("Ошибка: введите корректное число!")
```

Размышления: данное упражнение вводит концепции условных операторов (`if-elif-else`), обработки исключений (`try-except`) и валидации пользовательского ввода. Это критически важные навыки для создания надежных программ.

Упражнение 3. Анализатор числовых последовательностей

Создайте программу, которая анализирует последовательность чисел, введенных пользователем через запятую, и выводит статистику: количество чисел, их сумму, среднее значение, минимум и максимум.

```
# Анализатор числовых последовательностей
def analyze_sequence(numbers_str):
    """Анализирует строку с числами и возвращает статистику"""
    try:
        # Преобразуем строку в список чисел
        numbers = [float(x.strip()) for x in numbers_str.split(',')]

        if not numbers:
            return None

        stats = {
            'count': len(numbers),
            'sum': sum(numbers),
```

```
            'average': sum(numbers) / len(numbers),
            'min': min(numbers),
            'max': max(numbers)
        }

        return stats

    except ValueError:
        return None

# Основная программа
print("Анализатор числовых последовательностей")
print("Введите числа через запятую:")

user_input = input("Числа: ")
stats = analyze_sequence(user_input)

if stats:
    print(f"\n=== Статистика ===")
    print(f"Количество чисел: {stats['count']}")
    print(f"Сумма: {stats['sum']}")
    print(f"Среднее значение: {stats['average']:.2f}")
    print(f"Минимальное значение: {stats['min']}")
    print(f"Максимальное значение: {stats['max']}")
else:
    print("Ошибка при обработке данных. Проверьте формат ввода.")
```

Размышления: это упражнение развивает навыки работы со списками (lists), функциями и более сложной обработкой данных. Введение функции `analyze_sequence` демонстрирует принципы структурного программирования — разбиение сложной задачи на более простые, повторно используемые компоненты.

Упражнение 4. Генератор простых паролей

Напишите программу, которая генерирует простой пароль на основе введенной пользователем информации, комбинируя ее с числами и символами.

```
import random
import string

def generate_simple_password(base_word, length=8):
    """Генерирует пароль на основе базового слова"""
    # Обеспечиваем минимальную длину
    if len(base_word) > length - 2:
        base_word = base_word[:length-2]

    # Добавляем случайные цифры
    digits = ''.join(random.choices(string.digits, k=2))

    # Добавляем случайные символы, если нужно
    remaining_length = length - len(base_word) - len(digits)
```

```
    if remaining_length > 0:
        symbols = ''.join(random.choices('!@#$', k=remaining_length))
        password = base_word + digits + symbols
    else:
        password = base_word + digits

    return password

# Основная программа
print("Генератор простых паролей")
base = input("Введите базовое слово: ")
length = input("Желаемая длина пароля (по умолчанию 8): ")

try:
    length = int(length) if length else 8
    password = generate_simple_password(base, length)
    print(f"Сгенерированный пароль: {password}")
    print("Помните: используйте надежные пароли в реальных приложениях!")

except ValueError:
    print("Некорректная длина пароля!")
```

Размышления: данное упражнение вводит работу с модулем `random`, демонстрирует импорт внешних библиотек и показывает, как можно создавать полезные утилиты. Важно отметить предупреждение о безопасности — это формирует правильное отношение к вопросам информационной безопасности.

Заключительные размышления о первой главе

Мы прошли путь от понимания сути программирования до создания первых функциональных программ. Каждое упражнение развивает не только технические навыки, но и способность разбивать сложные задачи на простые компоненты — основу алгоритмического мышления.

Ключевые концепции, которые мы освоили:

- взаимодействие с пользователем через функции `input()` и `print()`;
- работа с различными типами данных и их преобразование;
- основы обработки ошибок и валидации данных;
- структурирование кода с помощью функций;
- различие между интерактивной разработкой и написанием скриптов.

Важно понимать, что программирование — это итеративный процесс. Не стремитесь написать идеальный код с первого раза. Экспериментируйте, делайте ошибки, анализируйте их и совершенствуйте свои решения. Каждая написанная строка кода приближает вас к мастерству программирования.

Глава 2. Язык данных и выражений

2.1. Типы данных: строительные блоки программ

Представьте, что вы архитектор, проектирующий здание. Прежде чем приступить к работе, вам необходимо знать, какие материалы доступны: кирпич, дерево, металл, стекло. Каждый материал имеет свои уникальные свойства и предназначение. Кирпич прочен и долговечен, но тяжел. Стекло пропускает свет, но хрупко. В программировании роль таких строительных материалов играют типы данных (data types).

Тип данных определяет не только то, какие значения может хранить переменная, но и то, какие операции можно с ними выполнять. Это фундаментальная концепция, которая влияет на то, как программа работает с информацией, сколько памяти она потребляет и насколько эффективно выполняются вычисления.

В Python существует несколько базовых типов данных, каждый из которых решает определенный класс задач:

- **целые числа (`int`)** — для точных вычислений с целыми значениями;
- **вещественные числа (`float`)** — для работы с дробями и научными вычислениями;
- **строки (`str`)** — для обработки текстовой информации;
- **булевы значения (`bool`)** — для логических операций и условий;
- **комплексные числа (`complex`)** — для математических расчетов в инженерии.

Понимание типов данных критически важно, потому что неправильный выбор типа может привести к неожиданным результатам. Например, деление целых чисел в некоторых языках дает целый результат, что может исказить финансовые расчеты.

```
# Демонстрация различных типов данных
age = 25                    # целое число (int)
price = 19.99               # вещественное число (float)
name = "Alice"              # строка (str)
is_student = True           # булево значение (bool)
coordinates = 3 + 4j        # комплексное число (complex)
# Проверка типа данных
print(type(age))            # <class 'int'>
print(type(price))          # <class 'float'>
print(type(name))           # <class 'str'>
print(type(is_student))     # <class 'bool'>
print(type(coordinates))    # <class 'complex'>
```

Особенность Python заключается в том, что он использует **динамическую типизацию (dynamic typing)**. Это означает, что тип переменной определяется автоматически на основе присвоенного значения. Вам не нужно заранее объявлять, что переменная будет числом или строкой — Python сам это поймет.

```
# Динамическая типизация в действии
data = 42               # data теперь int
print(type(data))       # <class 'int'>
```

```
data = "Hello"        # та же переменная теперь str
print(type(data))     # <class 'str'>
data = 3.14           # теперь она float
print(type(data))     # <class 'float'>
```

Однако эта гибкость требует особого внимания к контролю типов в критических участках программы. Хорошей практикой является явная проверка и преобразование типов когда это необходимо.

2.2. Переменные как именованные контейнеры

Если типы данных — это строительные материалы, то переменные (variables) — это именованные контейнеры для хранения этих материалов. Переменная не просто хранит значение; она создает абстракцию, позволяющую нам работать с данными через понятные имена вместо конкретных адресов в памяти.

Концепция переменной является одной из самых важных в программировании, поскольку она обеспечивает два критически важных аспекта.

1. **Читаемость кода** — `monthly_salary * 12` понятнее, чем `2500 * 12`.
2. **Переиспользование данных** — одно значение можно использовать в разных частях программы.

В Python создание переменной происходит через операцию присваивания. Важно понимать, что знак равенства (=) в программировании не означает математическое равенство. Это команда «присвоить значение переменной».

```
# Создание и использование переменных
student_name = "Maria"
current_age = 20
gpa = 3.85
is_enrolled = True
print(f"Студент {student_name}, возраст {current_age}")
print(f"Средний балл: {gpa}")
print(f"Зачислен: {is_enrolled}")
```

Правила именования переменных в Python следуют определенным конвенциям, которые делают код более профессиональным и читаемым:

- имена должны начинаться с буквы или символа подчеркивания;
- могут содержать буквы, цифры и символы подчеркивания;
- регистр имеет значение: `name` и `Name` — разные переменные;
- нельзя использовать зарезервированные слова Python.

```
# Правильные имена переменных
user_age = 25
_private_data = "secret"
counter1 = 0
MAX_SIZE = 100
# Неправильные имена (вызовут ошибку)
# 2names = "error"     # начинается с цифры
# class = "error"      # зарезервированное слово
# user-name = "error"  # содержит дефис
```

Важная особенность Python — переменные являются **ссылками на объекты**. Когда мы создаем переменную, мы не создаем копию данных, а создаем ссылку на объект в памяти. Это влияет на то, как ведут себя переменные при копировании и изменении.

```
# Демонстрация ссылочной природы переменных
original_list = [1, 2, 3]
copied_reference = original_list
# Изменяем список через одну переменную
copied_reference.append(4)
# Обе переменные показывают одинаковый результат
print(original_list)        # [1, 2, 3, 4]
print(copied_reference)     # [1, 2, 3, 4]
# Для простых типов (числа, строки) это работает по-другому
number1 = 10
number2 = number1
number2 = 20
print(number1)  # 10 (не изменилось)
print(number2)  # 20
```

2.3. Числа: целые, вещественные и их особенности

Числа в программировании — это не просто способ представления количественных данных. Это сложная система типов, каждый из которых оптимизирован для определенного класса задач. Понимание различий между типами чисел критически важно для написания корректных и эффективных программ.

Целые числа (`int`)

Целые числа в Python представляют собой точные математические целые без дробной части. В отличие от многих других языков программирования, Python не ограничивает размер целых чисел — они могут быть сколь угодно большими, ограниченными только доступной памятью.

```
# Примеры целых чисел
small_number = 42
large_number = 12345678901234567890123456789O
negative = -17
zero = 0
print(f"Маленькое число: {small_number}")
print(f"Большое число: {large_number}")
print(f"Отрицательное: {negative}")
# Проверка типа
print(type(small_number))  # <class 'int'>
```

Целые числа можно записывать в различных системах счисления, что особенно полезно при работе с низкоуровневым программированием:

```
# Различные системы счисления
decimal = 255        # десятичная
binary = 0b11111111  # двоичная (префикс 0b)
octal = 0o377        # восьмеричная (префикс 0o)
```

```
hexadecimal = 0xFF   # шестнадцатеричная (префикс 0x)
print(f"Все числа равны: {decimal == binary == octal == hexadecimal}")
```

Вещественные числа (`float`)

Вещественные числа предназначены для работы с дробными значениями и представлены в формате двойной точности IEEE 754. Это означает, что они хранятся в 64-битном формате и имеют ограниченную точность — приблизительно 15-17 значащих десятичных разрядов.

```
# Различные способы записи вещественных чисел
price = 19.99
scientific = 1.23e-4  # научная нотация: 1.23 × 10^(-4)
percentage = .85      # можно опустить ноль перед точкой
large_float = 1.5e6   # 1,500,000.0
print(f"Цена: {price}")
print(f"Научная нотация: {scientific}")
print(f"В процентах: {percentage}")
print(f"Большое число: {large_float}")
```

Критически важно понимать **ограничения точности** вещественных чисел. Это может привести к неожиданным результатам в вычислениях:

```
# Демонстрация проблем с точностью
result = 0.1 + 0.2
print(f"0.1 + 0.2 = {result}")              # 0.30000000000000004
print(f"Равно ли это 0.3? {result == 0.3}") # False
# Правильный способ сравнения вещественных чисел
tolerance = 1e-10
is_equal = abs(result - 0.3) < tolerance
print(f"С учетом погрешности: {is_equal}")  # True
```

Для финансовых расчетов, где точность критична, рекомендуется использовать модуль `decimal`:

```
from decimal import Decimal
# Точные десятичные вычисления
price1 = Decimal('19.99')
price2 = Decimal('10.01')
total = price1 + price2
print(f"Точная сумма: {total}")  # 30.00
```

Преобразование между типами чисел

Python предоставляет встроенные функции для преобразования между числовыми типами. Важно понимать, как происходят эти преобразования и когда может происходить потеря данных:

```
# Преобразование типов чисел
integer_num = 42
float_num = 3.14
string_num = "123"
# int → float (безопасно)
```

```
converted_float = float(integer_num)
print(f"int в float: {converted_float}")  # 42.0
# float → int (потеря дробной части)
converted_int = int(float_num)
print(f"float в int: {converted_int}")    # 3
# string → int (если возможно)
converted_from_string = int(string_num)
print(f"string в int: {converted_from_string}")  # 123
# Округление при преобразовании
print(f"int(3.7) = {int(3.7)}")    # 3 (усечение, не округление)
print(f"int(3.2) = {int(3.2)}")    # 3
print(f"round(3.7) = {round(3.7)}") # 4 (математическое округление)
```

Математические операции с числами

Понимание того, как Python выполняет арифметические операции, помогает предсказать результат вычислений и избежать ошибок:

```
# Основные арифметические операции
a = 17
b = 5
print(f"Сложение: {a} + {b} = {a + b}")           # 22
print(f"Вычитание: {a} - {b} = {a - b}")          # 12
print(f"Умножение: {a} * {b} = {a * b}")          # 85
print(f"Деление: {a} / {b} = {a / b}")            # 3.4 (всегда float)
print(f"Целочисленное деление: {a} // {b} = {a // b}")  # 3
print(f"Остаток от деления: {a} % {b} = {a % b}")       # 2
print(f"Возведение в степень: {a} ** {b} = {a ** b}")   # 1419857
# Смешанные операции
mixed_result = 10 + 3.14  # int + float → float
print(f"Смешанная операция: {mixed_result}, тип: {type(mixed_result)}")
```

Особенно важно понимать разницу между обычным делением (`/`) и целочисленным делением (`//`). Обычное деление всегда возвращает вещественное число, даже если оба операнда целые, а целочисленное деление возвращает наибольшее целое число, не превышающее результат деления.

2.4. Строки: работа с текстом в Python

Строки (strings) в программировании — это не просто последовательность символов. Это мощный инструмент для работы с текстовой информацией, который лежит в основе многих современных приложений: от веб-интерфейсов до систем обработки естественного языка. В эпоху цифровых технологий умение эффективно работать со строками определяет качество пользовательского опыта и функциональность программ.

Строка в Python — это неизменяемая (immutable) последовательность символов Unicode, что означает две важные вещи: во-первых, Python может работать с текстом на любом языке мира, во-вторых, после создания строки ее нельзя изменить, только создать новую.

Создание и базовые операции со строками

Строки в Python можно создавать несколькими способами, каждый из которых подходит для разных ситуаций:

```
# Различные способы создания строк
single_quotes = 'Hello, World!'
double_quotes = "Python Programming"
triple_quotes = """Многострочная строка

    может содержать переносы
    и специальные символы """
```

```
# Пустая строка
empty_string = ""
# Строка с кавычками внутри
quote_inside = "He said, 'Python is amazing!'"
another_quote = 'She replied, "I totally agree!"'
print(single_quotes)
print(double_quotes)
print(triple_quotes)
```

Экранирование символов (escaping) позволяет включать в строки специальные символы:

```
# Специальные символы и экранирование
newline_example = "Первая строка\nВторая строка"
tab_example = "Колонка1\tКолонка2\tКолонка3"
quote_example = "Он сказал: \"Это важно!\""
backslash_example = "Путь к файлу: C:\\Users\\Documents"
print(newline_example)
print(tab_example)
print(quote_example)
print(backslash_example)
# Сырые строки (raw strings) для избежания экранирования
raw_path = r"C:\Users\Documents\new_file.txt"
print(raw_path)
```

Индексация и срезы строк

Поскольку строка является последовательностью символов, к каждому символу можно обращаться по его позиции (индексу). Python использует нулевую индексацию — первый символ имеет индекс 0.

```
text = "Programming"
# Положительная индексация (слева направо)
print(f"Первый символ: {text[0]}")      # P
print(f"Пятый символ: {text[4]}")       # r
print(f"Последний символ: {text[10]}")  # g
# Отрицательная индексация (справа налево)
print(f"Последний символ: {text[-1]}")     # g
print(f"Предпоследний: {text[-2]}")        # n
print(f"Третий с конца: {text[-3]}")       # i
```

```
# Длина строки
print(f"Длина строки: {len(text)}")        # 11
```

Срезы (slicing) позволяют извлекать подстроки, используя синтаксис [начало:конец:шаг]:

```
programming = "Python Programming"
# Основные срезы
print(f"Первые 6 символов: {programming[:6]}")       # Python
print(f"С 7-го символа: {programming[7:]}")          # Programming
print(f"Символы с 7 по 11: {programming[7:12]}")     # Progr
print(f"Каждый второй символ: {programming[::2]}")   # Pto rgamn
# Обращение строки
reversed_string = programming[::-1]
print(f"Обращенная строка: {reversed_string}")       # gnimmargorP nohtyP
```

Методы строк для обработки текста

Python предоставляет богатый набор методов для работы со строками. Поскольку строки неизменяемы, все методы возвращают новые строки:

```
sample_text = "  Python Programming Language  "
# Очистка и форматирование
cleaned = sample_text.strip()            # удаление пробелов по краям
print(f"Очищенная: '{cleaned}'")
uppercase = cleaned.upper()              # преобразование в верхний регистр
lowercase = cleaned.lower()              # преобразование в нижний регистр
title_case = cleaned.title()             # заглавные буквы в словах
print(f"Верхний регистр: {uppercase}")
print(f"Нижний регистр: {lowercase}")
print(f"Заголовочный: {title_case}")
# Поиск и замена
position = cleaned.find("Programming")        # поиск подстроки
count = cleaned.count("g")                    # подсчет вхождений
replaced = cleaned.replace("Python", "Java")  # замена подстроки
print(f"Позиция 'Programming': {position}")
print(f"Количество 'g': {count}")
print(f"После замены: {replaced}")
```

Проверочные методы строк

Методы проверки возвращают булевы значения и часто используются для валидации данных:

```
# Примеры различных строк для проверки
numeric_string = "12345"
alpha_string = "Python"
mixed_string = "Python123"
space_string = "   "
# Проверки типа содержимого
print(f"'{numeric_string}' - только цифры: {numeric_string.isdigit()}")
print(f"'{alpha_string}' - только буквы: {alpha_string.isalpha()}")
```

```
print(f"'{mixed_string}' - буквы и цифры: {mixed_string.isalnum()}")
print(f"'{space_string}' - только пробелы: {space_string.isspace()}")
# Проверки регистра
print(f"'{alpha_string.upper()}' - верхний регистр: {alpha_string.upper().
isupper()}")
print(f"'{alpha_string.lower()}' - нижний регистр: {alpha_string.lower().
islower()}")
print(f"'Python Language' - заголовочный: {'Python Language'.istitle()}")
```

Форматирование строк

Современный Python предлагает несколько способов форматирования строк, каждый со своими преимуществами:

```
name = "Alice"
age = 25
score = 92.5
# f-строки (рекомендуемый способ в Python 3.6+)
f_string = f"Студент {name}, возраст {age}, оценка: {score:.1f}"
print(f_string)
# Метод format()
format_string = "Студент {}, возраст {}, оценка: {:.1f}".format(name, age,
score)
print(format_string)
# Именованные параметры в format()
named_format = "Студент {student}, возраст {years}, оценка: {grade:.1f}".
format(
    student=name, years=age, grade=score
)
print(named_format)
# Старый способ с оператором % (не рекомендуется)
old_format = "Студент %s, возраст %d, оценка: %.1f" % (name, age, score)
print(old_format)
```

2.5. Булевы значения: логика истины и лжи

Булевы значения (Boolean values) названы в честь математика Джорджа Буля и представляют собой простейший тип данных, который может принимать только два значения: `True` (истина) и `False` (ложь). Несмотря на кажущуюся простоту, булева логика является основой всех вычислений и принятия решений в программировании.

В Python булевы значения — это не просто константы, а полноценный тип данных со своими особенностями и правилами преобразования. Понимание того, как Python интерпретирует истинность различных значений, критически важно для написания корректных условий и логических выражений.

Основы булевых значений

```
# Создание булевых значений
is_student = True
is_graduated = False
```

```
has_license = True
print(f"Студент: {is_student}")
print(f"Закончил обучение: {is_graduated}")
print(f"Есть лицензия: {has_license}")
# Проверка типа
print(f"Тип True: {type(True)}")        # <class 'bool'>
print(f"Тип False: {type(False)}")      # <class 'bool'>
# Булевы значения являются подклассом int
print(f"True как число: {int(True)}")   # 1
print(f"False как число: {int(False)}") # 0
```

Истинность различных типов данных

Python имеет четкие правила определения истинности для всех типов данных. Это концепция называется truthiness — способность значения быть интерпретированным как истинное или ложное в логическом контексте:

```
# Ложные значения (Falsy values)
print("=== ЛОЖНЫЕ ЗНАЧЕНИЯ ===")
print(f"False: {bool(False)}")
print(f"None: {bool(None)}")
print(f"Ноль (int): {bool(0)}")
print(f"Ноль (float): {bool(0.0)}")
print(f"Пустая строка: {bool('')}")
print(f"Пустой список: {bool([])}")
print(f"Пустой словарь: {bool({})}")
# Истинные значения (Truthy values)
print("\n=== ИСТИННЫЕ ЗНАЧЕНИЯ ===")
print(f"True: {bool(True)}")
print(f"Любое ненулевое число: {bool(42)}")
print(f"Непустая строка: {bool('Python')}")
print(f"Непустой список: {bool([1, 2, 3])}")
print(f"Непустой словарь: {bool({'key': 'value'})}")
```

Логические операторы

Python предоставляет три основных логических оператора для работы с булевыми значениями:

```
# Примеры для демонстрации логических операторов
is_sunny = True
is_warm = False
has_umbrella = True
# Оператор AND (и) - истина, только если оба условия истинны
good_weather = is_sunny and is_warm
print(f"Хорошая погода (солнечно И тепло): {good_weather}")
# Оператор OR (или) - истина, если хотя бы одно условие истинно
can_go_outside = is_sunny or has_umbrella
print(f"Можно выйти (солнечно ИЛИ есть зонт): {can_go_outside}")
# Оператор NOT (не) - инверсия булевого значения
is_cloudy = not is_sunny
```

```
print(f"Облачно (НЕ солнечно): {is_cloudy}")
# Сложные логические выражения
should_take_umbrella = (not is_sunny) and (not has_umbrella)
print(f"Нужно взять зонт: {should_take_umbrella}")
```

Операторы сравнения

Операторы сравнения всегда возвращают булевы значения и являются основой для создания условий:

```
# Числовые сравнения
x, y = 10, 20
print(f"{x} == {y}: {x == y}")  # равно
print(f"{x} != {y}: {x != y}")  # не равно
print(f"{x} < {y}: {x < y}")    # меньше
print(f"{x} > {y}: {x > y}")    # больше
print(f"{x} <= {y}: {x <= y}")  # меньше или равно
print(f"{x} >= {y}: {x >= y}")  # больше или равно
# Сравнение строк (лексикографическое)
name1, name2 = "Alice", "Bob"
print(f"'{name1}' < '{name2}': {name1 < name2}")  # True (A идет перед B)
# Сравнение с учетом регистра
print(f"'apple' < 'Banana': {'apple' < 'Banana'}")  # False (заглавные буквы "меньше")
# Цепочки сравнений (удобная особенность Python)
age = 25
is_adult = 18 <= age <= 65
print(f"Возраст {age} — взрослый (18-65): {is_adult}")
```

Особенности работы с булевыми значениями

Важно понимать некоторые тонкости работы с булевыми значениями в Python:

```
# Короткое замыкание (short-circuit evaluation)
def expensive_operation():
    print("Выполняется дорогая операция!")
    return True
# В выражении AND, если первое условие False, второе не проверяется
result1 = False and expensive_operation()  # expensive_operation() не вызывается
print(f"Результат 1: {result1}")
# В выражении OR, если первое условие True, второе не проверяется
result2 = True or expensive_operation()    # expensive_operation() не вызывается
print(f"Результат 2: {result2}")
# Использование булевых значений в арифметике
votes_for = 15
votes_against = 8
has_majority = votes_for > votes_against
# True/False можно использовать как 1/0
```

```
total_decisions = int(has_majority) + int(not has_majority)  # всегда рав-
но 1
print(f"Принятых решений: {total_decisions}")
```

2.6. Операторы и приоритет операций

Операторы (operators) — это символы или ключевые слова, которые сообщают интерпретатору Python, какую операцию нужно выполнить над операндами. Понимание приоритета операций критически важно для написания корректных выражений, поскольку неправильный порядок вычислений может привести к совершенно неожиданным результатам.

В математике мы изучаем, что сначала выполняется умножение, а потом сложение. В программировании правила более сложные и включают множество различных типов операторов. Python следует четко определенной иерархии приоритетов, которая определяет порядок выполнения операций в сложных выражениях.

Арифметические операторы

Арифметические операторы выполняют математические вычисления и имеют приоритеты, аналогичные математическим правилам:

```
# Демонстрация приоритета арифметических операторов
result1 = 2 + 3 * 4       # Сначала умножение: 2 + 12 = 14
result2 = (2 + 3) * 4     # Скобки изменяют порядок: 5 * 4 = 20
result3 = 2 ** 3 ** 2     # Возведение в степень правоассоциативно: 2 ** (3
** 2) = 2 ** 9 = 512
print(f"2 + 3 * 4 = {result1}")
print(f"(2 + 3) * 4 = {result2}")
print(f"2 ** 3 ** 2 = {result3}")
# Деление и его особенности
division_float = 15 / 4        # Обычное деление: 3.75
division_integer = 15 // 4     # Целочисленное деление: 3
remainder = 15 % 4             # Остаток от деления: 3
print(f"15 / 4 = {division_float}")
print(f"15 // 4 = {division_integer}")
print(f"15 % 4 = {remainder}")
# Проверка: делимое = частное * делитель + остаток
verification = division_integer * 4 + remainder
print(f"Проверка: {division_integer} * 4 + {remainder} = {verification}")
```

Операторы сравнения и их цепочки

Операторы сравнения имеют более низкий приоритет, чем арифметические, но могут образовывать цепочки:

```
# Приоритет сравнений ниже арифметических операций
x, y, z = 5, 10, 15
# Эти выражения эквивалентны благодаря приоритету
comparison1 = x + 5 < y * 2    # (x + 5) < (y * 2) = 10 < 20 = True
comparison2 = (x + 5) < (y * 2)
print(f"x + 5 < y * 2: {comparison1}")
```

```
print(f"Эквивалентное: {comparison2}")
# Цепочки сравнений — уникальная особенность Python
age = 25
is_working_age = 18 <= age <= 65
print(f"Рабочий возраст (18-65): {is_working_age}")
# Это эквивалентно, но менее читаемо:
is_working_age_verbose = (18 <= age) and (age <= 65)
print(f"Подробная запись: {is_working_age_verbose}")
# Цепочки могут быть сколь угодно длинными
values = [1, 2, 3, 4, 5]
is_ascending = values[0] < values[1] < values[2] < values[3] < values[4]
print(f"Возрастающая последовательность: {is_ascending}")
```

Логические операторы и короткое замыкание

Логические операторы имеют свой порядок приоритета и используют механизм короткого замыкания:

```
# Приоритет логических операторов: not > and > or
def test_function(name, value):
    print(f"Вызвана функция {name} с значением {value}")
    return value
# Демонстрация короткого замыкания
print("=== Тест короткого замыкания для AND ===")
result_and = test_function("first", False) and test_function("second", True)
print(f"Результат AND: {result_and}")  # second не вызывается
print("\n=== Тест короткого замыкания для OR ===")
result_or = test_function("first", True) or test_function("second", False)
print(f"Результат OR: {result_or}")    # second не вызывается
# Сложные логические выражения
temperature = 25
humidity = 60
is_sunny = True
# Приоритет: not > and > or
comfortable = temperature > 20 and humidity < 70 or is_sunny and temperature
> 15
# Эквивалентно: ((temperature > 20) and (humidity < 70)) or (is_sunny and
(temperature > 15))
print(f"\nКомфортные условия: {comfortable}")
```

Операторы присваивания

Python предоставляет составные операторы присваивания для более компактной записи:

```
# Базовое присваивание
counter = 0
balance = 1000.0
# Составные операторы присваивания
counter += 1        # эквивалентно: counter = counter + 1
balance -= 50.5     # эквивалентно: balance = balance - 50.5
```

```
counter *= 2        # эквивалентно: counter = counter * 2
balance /= 4        # эквивалентно: balance = balance / 4
print(f"Counter: {counter}")    # 2
print(f"Balance: {balance}")    # 237.375
# Составные операторы со строками и списками
message = "Hello"
message += " World!"            # конкатенация строк
print(f"Message: {message}")    # Hello World!
numbers = [1, 2, 3]
numbers += [4, 5]               # расширение списка
print(f"Numbers: {numbers}")    # [1, 2, 3, 4, 5]
```

Полная таблица приоритетов операторов

Понимание полной иерархии приоритетов помогает читать и писать сложные выражения:

```
# Пример сложного выражения с различными операторами
x, y, z = 10, 5, 2
is_positive = True
# Разбор выражения по приоритетам:
# 1. Скобки: (y + z)
# 2. Возведение в степень: z ** 2
# 3. Умножение: x * (y + z)
# 4. Сравнение: result > z ** 2
# 5. Логическое AND
complex_expression = is_positive and x * (y + z) > z ** 2
print(f"Сложное выражение: {complex_expression}")
# Пошаговый разбор для понимания:
step1 = y + z                   # 5 + 2 = 7
step2 = z ** 2                  # 2 ** 2 = 4
step3 = x * step1               # 10 * 7 = 70
step4 = step3 > step2           # 70 > 4 = True
step5 = is_positive and step4   # True and True = True
print(f"Пошаговый разбор: {step5}")
# Использование скобок для ясности
clear_expression = is_positive and ((x * (y + z)) > (z ** 2))
print(f"С явными скобками: {clear_expression}")
```

Рекомендация по стилю кода: даже когда вы знаете приоритет операторов, использование скобок для группировки сложных выражений делает код более читаемым и снижает вероятность ошибок. Код должен быть написан не только для компьютера, но и для других программистов, которые будут его читать и поддерживать.

2.7. Ввод данных от пользователя

Интерактивность — одно из ключевых качеств современных программ. Способность программы получать данные от пользователя, обрабатывать их и предоставлять результат превращает статичный код в живой инструмент решения задач. В Python функция `input()` является основным механизмом взаимодействия с пользователем через консоль.

Функция `input()` всегда возвращает строку, независимо от того, что вводит пользователь. Это важнейшая особенность, которую необходимо учитывать при обработке пользовательского ввода. Понимание этого принципа поможет избежать множества ошибок в программах.

Базовое использование функции `input()`

```
# Простейший пример ввода данных
name = input("Введите ваше имя: ")
print(f"Привет, {name}!")
# Ввод всегда возвращает строку
age_string = input("Введите ваш возраст: ")
print(f"Вы ввели: '{age_string}', тип: {type(age_string)}")
# Для числовых операций необходимо преобразование типа
age = int(age_string)
next_year_age = age + 1
print(f"В следующем году вам будет {next_year_age} лет")
```

Преобразование типов при вводе

Поскольку `input()` всегда возвращает строку, для работы с числовыми данными требуется явное преобразование типов:

```
# Ввод целого числа
try:
    number = int(input("Введите целое число: "))
    squared = number ** 2
    print(f"Квадрат числа {number} равен {squared}")
except ValueError:
    print("Ошибка: введено некорректное целое число")
# Ввод вещественного числа
try:
    price = float(input("Введите цену товара: "))
    with_tax = price * 1.2  # добавляем 20% налога
    print(f"Цена с налогом: {with_tax:.2f}")
except ValueError:
    print("Ошибка: введена некорректная цена")
# Ввод булевого значения
response = input("Согласны ли вы? (да/нет): ").lower()
agreement = response in ['да', 'yes', 'y', '1', 'true']
print(f"Ваше согласие: {agreement}")
```

Создание надежного ввода с проверкой

Профессиональные программы должны корректно обрабатывать некорректный ввод пользователя:

```
def get_integer(prompt, min_value=None, max_value=None):
    """Функция для безопасного ввода целого числа с проверкой диапазона"""
    while True:
        try:
```

```
            value = int(input(prompt))

            # Проверка минимального значения
            if min_value is not None and value < min_value:
                print(f"Значение должно быть не менее {min_value}")
                continue

            # Проверка максимального значения
            if max_value is not None and value > max_value:
                print(f"Значение должно быть не более {max_value}")
                continue

            return value

        except ValueError:
            print("Пожалуйста, введите корректное целое число")
# Использование надежного ввода
age = get_integer("Введите ваш возраст (0-120): ", min_value=0, max_
value=120)
grade = get_integer("Введите оценку (1-5): ", min_value=1, max_value=5)
print(f"Возраст: {age}, Оценка: {grade}")
```

Ввод сложных данных

Часто требуется ввести несколько значений или более сложные структуры данных:

```
# Ввод нескольких значений через пробел
coordinates_input = input("Введите координаты (x y): ")
try:
    x, y = map(float, coordinates_input.split())
    distance = (x**2 + y**2) ** 0.5
    print(f"Расстояние от начала координат: {distance:.2f}")
except ValueError:
    print("Ошибка: введите два числа через пробел")
# Ввод списка значений
numbers_input = input("Введите числа через запятую: ")
try:
    numbers = [float(x.strip()) for x in numbers_input.split(',')]
    average = sum(numbers) / len(numbers)
    print(f"Среднее значение: {average:.2f}")
    print(f"Количество чисел: {len(numbers)}")
except (ValueError, ZeroDivisionError):
    print("Ошибка: проверьте корректность введенных данных")
# Ввод данных до специального символа
print("Введите строки текста (пустая строка для завершения):")
lines = []
while True:
    line = input()
    if line == "":
        break
```

```
    lines.append(line)
print(f"\nВы ввели {len(lines)} строк:")
for i, line in enumerate(lines, 1):
    print(f"{i}: {line}")
```

2.8. Форматирование вывода: красота в деталях

Качественное форматирование вывода — это не просто эстетический вопрос. Правильно представленная информация улучшает пользовательский опыт, снижает количество ошибок в интерпретации данных и демонстрирует профессионализм разработчика. В эпоху пользовательских интерфейсов даже консольные программы должны представлять данные ясно и структурированно.

f-строки: современный стандарт форматирования

f-строки (formatted string literals), введенные в Python 3.6, представляют собой наиболее читаемый и эффективный способ форматирования строк:

```
# Базовое использование f-строк
name = "Alice"
age = 25
balance = 1234.567
basic_info = f"Имя: {name}, Возраст: {age}, Баланс: {balance}"
print(basic_info)
# Выполнение выражений внутри f-строк
hours_worked = 40
hourly_rate = 25.50
weekly_salary = f"Зарплата за неделю: ${hours_worked * hourly_rate:.2f}"
print(weekly_salary)
# Вызов методов и функций внутри f-строк
text = "python programming"
formatted = f"Заголовок: {text.title()}, Длина: {len(text)} символов"
print(formatted)
```

Спецификаторы формата для чисел

f-строки поддерживают мощную систему спецификаторов формата для точного контроля представления данных:

```
import math
# Форматирование вещественных чисел
pi = math.pi
price = 1234.5678
print(f"Число π с двумя знаками: {pi:.2f}")          # 3.14
print(f"Число π с пятью знаками: {pi:.5f}")          # 3.14159
print(f"Цена с округлением: {price:.2f}")            # 1234.57
# Экспоненциальная запись
large_number = 1234567.89
print(f"Научная нотация: {large_number:.2e}")        # 1.23e+06
print(f"Фиксированная точка: {large_number:.2f}")    # 1234567.89
# Процентное представление
```

```
success_rate = 0.856
print(f"Процент успеха: {success_rate:.1%}")        # 85.6%
# Форматирование целых чисел
population = 1234567
print(f"Население: {population:,}")                 # 1,234,567 (разделите-
ли тысяч)
print(f"Двоичное: {population:b}")                  # 100101101011010000111
print(f"Шестнадцатеричное: {population:x}")         # 12d687
```

Выравнивание и заполнение

Контроль над выравниванием текста особенно важен при создании таблиц и отчетов:

```
# Выравнивание строк
name1, name2, name3 = "Alice", "Bob", "Catherine"
score1, score2, score3 = 95.5, 87.2, 92.8
print("=== ВЫРАВНИВАНИЕ ПО ЛЕВОМУ КРАЮ ===")
print(f"{name1:<10} | {score1:>6.1f}")
print(f"{name2:<10} | {score2:>6.1f}")
print(f"{name3:<10} | {score3:>6.1f}")
print("\n=== ВЫРАВНИВАНИЕ ПО ЦЕНТРУ ===")
print(f"{name1:^15} | {score1:^8.1f}")
print(f"{name2:^15} | {score2:^8.1f}")
print(f"{name3:^15} | {score3:^8.1f}")
print("\n=== ЗАПОЛНЕНИЕ СИМВОЛАМИ ===")
print(f"{name1:.<20} {score1:.1f}")
print(f"{name2:.<20} {score2:.1f}")
print(f"{name3:.<20} {score3:.1f}")
# Создание красивых заголовков
title = "ОТЧЕТ ПО ПРОДАЖАМ"
width = 50
print(f"{title:=^{width}}")
print(f"{'Товар':<20} {'Количество':>10} {'Сумма':>15}")
print("-" * width)
```

Форматирование дат и времени

Работа с датами требует особого внимания к форматированию:

```
from datetime import datetime, date
# Текущие дата и время
now = datetime.now()
today = date.today()
# Различные форматы даты
print(f"Полная дата: {now:%A, %B %d, %Y}")                    # Monday,
January 15, 2024
print(f"Краткая дата: {today:%d.%m.%Y}")                     # 15.01.2024
print(f"Время: {now:%H:%M:%S}")                              # 14:30:45
print(f"12-часовой формат: {now:%I:%M %p}")                  # 02:30 PM
# Локализованное форматирование
```

```
print(f"ISO формат: {now:%Y-%m-%d %H:%M:%S}")                    # 2024-01-15
14:30:45
# Пользовательские форматы для отчетов
report_date = f"Отчет сгенерирован: {now:%d %B %Y г. в %H:%M}"
print(report_date)
```

Создание структурированного вывода

Профессиональные программы часто требуют создания таблиц, отчетов и структурированного вывода:

```
# Данные для демонстрации
products = [
    ("Ноутбук", 1299.99, 5),
    ("Мышь", 29.99, 25),
    ("Клавиатура", 79.99, 12),
    ("Монитор", 299.99, 8)
]
# Создание таблицы товаров
header = f"{'Товар':<15} {'Цена':>10} {'Количество':>12} {'Стоимость':>12}"
separator = "-" * len(header)
print(header)
print(separator)
total_value = 0
for product, price, quantity in products:
    item_total = price * quantity
    total_value += item_total

  print(f"{product:<15} {price:>8.2f}$ {quantity:>10} {item_total:>10.2f}$")
print(separator)
print(f"{'ИТОГО:':<15} {'':<10} {'':<12} {total_value:>10.2f}$")
# Создание чека
print("\n" + "="*40)
print(f"{'ЧЕК':^40}")
print("="*40)
print(f"Дата: {datetime.now():%d.%m.%Y %H:%M}")
print("-"*40)
for i, (product, price, quantity) in enumerate(products, 1):
       print(f"{i}. {product:<20} {quantity} x {price:.2f}$ = {price *
quantity:.2f}$")
print("-"*40)
print(f"{'К оплате:':<30} {total_value:.2f}$")
print("="*40)
```

Многострочное форматирование

Для сложных отчетов и документов часто требуется многострочное форматирование:

```
# Данные студента
student_data = {
```

```
    'name': 'Мария Иванова',
    'group': 'ИТ-301',
    'subjects': [
        ('Математика', 4.5),
        ('Программирование', 5.0),
        ('Физика', 4.0),
        ('Английский', 4.2)
    ]
}
# Создание академической справки
report = f"""
{'АКАДЕМИЧЕСКАЯ СПРАВКА':=^50}
Студент: {student_data['name']}
Группа:  {student_data['group']}
Дата:    {datetime.now():%d.%m.%Y}
{'УСПЕВАЕМОСТЬ':^50}
{'-'*50}
"""
total_score = 0
subject_count = len(student_data['subjects'])
for subject, grade in student_data['subjects']:
    report += f"{subject:<25} {grade:>8.1f}\n"
    total_score += grade
average = total_score / subject_count
report += f"""
{'-'*50}
{'Средний балл:':<25} {average:>8.2f}
{'Справка выдана':<30} {datetime.now():%d.%m.%Y}
{'='*50}
"""
print(report)
```

2.9. Упражнения для закрепления

Теоретические знания становятся практическими навыками только через решение задач. Представленные упражнения организованы по принципу возрастающей сложности — от базовых операций с типами данных до создания комплексных интерактивных программ. Каждое упражнение направлено на закрепление конкретных концепций и развитие алгоритмического мышления.

Упражнение 1. Калькулятор типов данных

Создайте программу, которая демонстрирует работу с различными типами данных:

```
# Задание: создайте переменные разных типов и выведите информацию о них
# Пример решения:
def data_type_explorer():
    """Исследователь типов данных"""

    # Создание переменных разных типов
```

```
    integer_var = 42
    float_var = 3.14159
    string_var = "Python Programming"
    boolean_var = True
    complex_var = 2 + 3j

    variables = [
        ("Целое число", integer_var),
        ("Вещественное число", float_var),
        ("Строка", string_var),
        ("Булево значение", boolean_var),
        ("Комплексное число", complex_var)
    ]

    print("=== АНАЛИЗ ТИПОВ ДАННЫХ ===")
    for description, var in variables:
        print(f"{description:<20} | {var:<15} | {type(var).__name__}")

    # Демонстрация операций
    print(f"\nАрифметические операции:")
     print(f"{integer_var} + {float_var} = {integer_var + float_var} (тип:
{type(integer_var + float_var).__name__})")

    # Строковые операции
    print(f"\nСтроковые операции:")
    print(f"Длина строки '{string_var}': {len(string_var)}")
    print(f"Верхний регистр: {string_var.upper()}")
# Вызов функции
data_type_explorer()
```

Упражнение 2. Конвертер единиц измерения

Создайте интерактивную программу для конвертации различных единиц измерения:

```
def unit_converter():
    """Интерактивный конвертер единиц измерения"""

    print("=== КОНВЕРТЕР ЕДИНИЦ ИЗМЕРЕНИЯ ===")
    print("1. Температура (Цельсий ↔ Фаренгейт)")
    print("2. Длина (Метры ↔ Футы)")
    print("3. Вес (Килограммы ↔ Фунты)")

    choice = input("Выберите тип конвертации (1-3): ")

    if choice == '1':
        temp = float(input("Введите температуру: "))
        scale = input("Из какой шкалы конвертируете (C/F)? ").upper()

        if scale == 'C':
            fahrenheit = temp * 9/5 + 32
```

```
            print(f"{temp}°C = {fahrenheit:.1f}°F")
        elif scale == 'F':
            celsius = (temp - 32) * 5/9
            print(f"{temp}°F = {celsius:.1f}°C")

    elif choice == '2':
        length = float(input("Введите длину: "))
        unit = input("Из какой единицы конвертируете (M/F)? ").upper()

        if unit == 'M':
            feet = length * 3.28084
            print(f"{length} м = {feet:.2f} футов")
        elif unit == 'F':
            meters = length / 3.28084
            print(f"{length} футов = {meters:.2f} м")
# Раскомментируйте для запуска:
# unit_converter()
```

Упражнение 3. Анализатор текста

Создайте программу для анализа текстовой информации:

```
def text_analyzer():
    """Комплексный анализатор текста"""

    text = input("Введите текст для анализа: ")

    # Базовая статистика
    char_count = len(text)
    word_count = len(text.split())
    sentence_count = text.count('.') + text.count('!') + text.count('?')

    # Анализ символов
    letters = sum(1 for char in text if char.isalpha())
    digits = sum(1 for char in text if char.isdigit())
    spaces = sum(1 for char in text if char.isspace())

    # Создание отчета
    report = f"""
    {'АНАЛИЗ ТЕКСТА':=^50}

    Исходный текст: {text[:30]}{'...' if len(text) > 30 else ''}

    {'БАЗОВАЯ СТАТИСТИКА':-^50}
    Символов:      {char_count:>10}
    Слов:          {word_count:>10}
    Предложений:   {sentence_count:>10}

    {'ДЕТАЛЬНЫЙ АНАЛИЗ':-^50}
    Букв:          {letters:>10}
    Цифр:          {digits:>10}
    Пробелов:      {spaces:>10}
```

```
    {'ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ':-^50}
    Средняя длина слова: {letters/word_count if word_count > 0 else 0:.1f}
     Процент букв:          {letters/char_count*100 if char_count > 0 else
0:.1f}%
    """

    print(report)
# Раскомментируйте для запуска:
# text_analyzer()
```

Упражнение 4. Система оценки знаний

Разработайте интерактивную систему для оценки результатов тестирования:

```
def grading_system():
    """Система подсчета и анализа оценок"""

    print("=== СИСТЕМА ОЦЕНКИ ЗНАНИЙ ===")

    # Ввод данных студента
    student_name = input("Введите имя студента: ")
    subject = input("Введите предмет: ")

    print(f"\nВведите оценки для {student_name} по предмету '{subject}'")
    print("(введите пустую строку для завершения)")

    grades = []
    while True:
        grade_input = input(f"Оценка {len(grades) + 1}: ")
        if grade_input == "":
            break

        try:
            grade = float(grade_input)
            if 0 <= grade <= 5:
                grades.append(grade)
            else:
                print("Оценка должна быть от 0 до 5")
        except ValueError:
            print("Пожалуйста, введите корректное число")

    if not grades:
        print("Оценки не введены!")
        return

    # Расчет статистики
    average = sum(grades) / len(grades)
    max_grade = max(grades)
    min_grade = min(grades)
    # Определение уровня успеваемости
    if average >= 4.5:
        performance = "Отличная"
```

```
    elif average >= 3.5:
        performance = "Хорошая"
    elif average >= 2.5:
        performance = "Удовлетворительная"
    else:
        performance = "Неудовлетворительная"

    # Создание отчета
    print(f"\n{'АКАДЕМИЧЕСКИЙ ОТЧЕТ':=^60}")
    print(f"Студент: {student_name}")
    print(f"Предмет: {subject}")
    print(f"Дата: {datetime.now():%d.%m.%Y}")
    print("-" * 60)

    print(f"Оценки: {', '.join(f'{g:.1f}' for g in grades)}")
    print(f"Количество оценок: {len(grades)}")
    print(f"Средний балл: {average:.2f}")
    print(f"Максимальная оценка: {max_grade:.1f}")
    print(f"Минимальная оценка: {min_grade:.1f}")
    print(f"Успеваемость: {performance}")

    # Рекомендации
    print("\nРекомендации:")
    if average < 3.0:
        print("• Необходимо дополнительное изучение материала")
        print("• Рекомендуется консультация с преподавателем")
    elif average < 4.0:
        print("• Хорошие базовые знания, есть потенциал для улучшения")
        print("• Уделите внимание сложным темам")
    else:
        print("• Отличные знания предмета!")
        print("• Продолжайте в том же духе")
# Раскомментируйте для запуска:
# grading_system()
```

Упражнение 5. Личный финансовый помощник

Создайте комплексную программу для управления личными финансами:

```
def financial_assistant():
    """Персональный финансовый помощник"""

    print("=== ЛИЧНЫЙ ФИНАНСОВЫЙ ПОМОЩНИК ===")

    # Ввод базовых данных
    monthly_income = float(input("Ежемесячный доход: "))

    print("\nВведите ежемесячные расходы по категориям:")
    categories = ['Жилье', 'Питание', 'Транспорт', 'Развлечения', 'Прочие']
    expenses = {}
```

```
    total_expenses = 0
    for category in categories:
        amount = float(input(f"{category}: "))
        expenses[category] = amount
        total_expenses += amount

    # Расчеты
    savings = monthly_income - total_expenses
    savings_rate = (savings / monthly_income) * 100 if monthly_income > 0 else 0

    # Анализ бюджета
    print(f"\n{'ФИНАНСОВЫЙ АНАЛИЗ':=^60}")
    print(f"{'Доходы:':<20} {monthly_income:>15.2f} ₽")
    print(f"{'Расходы:':<20} {total_expenses:>15.2f} ₽")
    print("-" * 40)
    print(f"{'Остаток:':<20} {savings:>15.2f} ₽")
    print(f"{'Норма сбережений:':<20} {savings_rate:>12.1f}%")

    print(f"\n{'СТРУКТУРА РАСХОДОВ':-^40}")
    for category, amount in expenses.items():
        percentage = (amount / total_expenses) * 100 if total_expenses > 0
else 0
        print(f"{category:<15} {amount:>8.2f} ₽ ({percentage:>5.1f}%)")

    # Рекомендации
    print(f"\n{'РЕКОМЕНДАЦИИ':-^40}")
    if savings_rate >= 20:
        print("✓ Отличная норма сбережений!")
    elif savings_rate >= 10:
        print("• Хорошая норма сбережений, можно улучшить")
    elif savings_rate > 0:
        print("⚠ Низкая норма сбережений, пересмотрите расходы")
    else:
        print("⚠ КРИТИЧНО: Расходы превышают доходы!")

    # Прогноз накоплений
    if savings > 0:
        print(f"\nПрогноз накоплений:")
        periods = [6, 12, 24, 60]  # месяцы
        for months in periods:
            total_savings = savings * months
            years = months // 12
            remaining_months = months % 12

            period_text = f"{years} г." if years > 0 else ""
            if remaining_months > 0:
                period_text += f" {remaining_months} мес."

            print(f"За {period_text:<8} {total_savings:>10.2f} ₽")
# Раскомментируйте для запуска:
# financial_assistant()
```

Рекомендации по выполнению упражнений

Методика решения задач

1. **Понимание задачи** — внимательно прочитайте условие и определите, какие типы данных и операции потребуются.
2. **Планирование решения** — разбейте задачу на более мелкие подзадачи.
3. **Пошаговая реализация** — начните с простейших частей и постепенно усложняйте.
4. **Тестирование** — проверьте программу на различных входных данных.
5. **Рефакторинг** — улучшите код, сделайте его более читаемым и эффективным.

Дополнительные задания для самостоятельной работы

1. Модифицируйте калькулятор типов данных для работы с комплексными числами.
2. Добавьте в конвертер единиц новые типы измерений (объем, площадь, скорость).
3. Расширьте анализатор текста функциями поиска самых частых слов.
4. Создайте систему для сравнения успеваемости нескольких студентов.
5. Добавьте в финансовый помощник функцию планирования крупных покупок.
6. Реализуйте сохранение отчетов в текстовые файлы.

Эти упражнения не только закрепляют изученный материал, но и готовят к решению реальных задач программирования. Помните: программирование — это навык, который развивается только через практику. Каждая решенная задача приближает вас к мастерству в создании качественных программ.

Заключение

Во второй главе мы изучили фундаментальные концепции работы с данными в Python. Овладение типами данных, переменными, операторами и форматированием вывода создает прочную основу для дальнейшего изучения программирования. Эти знания будут использоваться в каждой программе, которую вы напишете.

Особое внимание следует уделить пониманию того, как Python интерпретирует различные типы данных и как происходят автоматические преобразования типов. Это поможет избежать множества ошибок в будущем и писать более надежный код.

Глава 3. Искусство принятия решений

Программирование, как и сама жизнь, состоит из постоянных решений. Каждый день мы принимаем тысячи больших и малых решений: что надеть, что съесть на завтрак, каким путем добраться на работу. Компьютерные программы сталкиваются с аналогичной необходимостью — они должны уметь анализировать ситуацию и выбирать соответствующий курс действий.

В предыдущих главах мы изучили основные типы данных и научились создавать простые линейные программы, которые выполняются строго последовательно, от первой строки к последней. Однако реальные программы редко работают столь прямолинейно. Они должны реагировать на различные ситуации, обрабатывать разные варианты входных данных и принимать решения на основе текущего состояния.

Условные конструкции (conditional statements) — это фундаментальный инструмент, который позволяет программам «думать» и адаптироваться к обстоятельствам. Они превращают статичную последовательность команд в динамичную систему, способную к анализу и принятию решений.

3.1. Условные конструкции: развилки в программе

Представьте, что вы идете по дороге и доходите до развилки. В зависимости от того, куда вам нужно попасть, вы выбираете левый или правый путь. Условные конструкции в программировании работают точно так же — они создают развилки в логике программы, где выполнение может пойти по одному из нескольких возможных путей.

В Python основной условной конструкцией является оператор `if` (если). Его базовая форма выглядит следующим образом:

```
if условие:
    # код, который выполняется, если условие истинно
    действие1
    действие2
```

Ключевое слово `if` сообщает интерпретатору Python, что далее следует условие для проверки. Если это условие истинно (`True`), выполняется блок кода, расположенный с отступом после двоеточия. Если условие ложно (`False`), этот блок пропускается.

Рассмотрим простой практический пример:

```
temperature = float(input("Введите температуру воздуха: "))
if temperature > 25:
    print("На улице жарко, оденьтесь легко")
    print("Не забудьте взять воду")
```

В этом примере программа запрашивает у пользователя температуру и, если она превышает 25 градусов, выводит соответствующие рекомендации. Еще раз обратите внимание на отступы — они не просто украшение кода, а важная часть синтаксиса Python, определяющая, какие строки входят в условный блок.

Часто нам нужно предусмотреть альтернативный вариант действий — что делать, если условие не выполняется. Для этого используется конструкция `if-else`:

```
age = int(input("Введите ваш возраст: "))
if age >= 18:
    print("Вы совершеннолетний")
    print("Можете участвовать в голосовании")
else:
    print("Вы несовершеннолетний")
    print("До совершеннолетия осталось лет:", 18 - age)
```

Конструкция `else` (иначе) позволяет определить блок кода, который выполнится в случае, если основное условие окажется ложным. Это создает бинарную развилку: программа обязательно пойдет по одному из двух путей.

Иногда нужно проверить несколько различных условий подряд. Для этого существует конструкция `elif` (сокращение от "`else if`" — «иначе если»):

```
grade = float(input("Введите вашу оценку (1-100): "))
if grade >= 90:
    print("Отлично! У вас оценка A")
elif grade >= 80:
    print("Хорошо! У вас оценка B")
elif grade >= 70:
    print("Удовлетворительно. У вас оценка C")
elif grade >= 60:
    print("Нужно подтянуться. У вас оценка D")
else:
    print("Неудовлетворительно. У вас оценка F")
```

Важно понимать, что условия в цепочке `if-elif-else` проверяются последовательно, и как только одно из них оказывается истинным, соответствующий блок выполняется, а все остальные пропускаются. Это означает, что порядок условий имеет значение.

Условия в Python могут быть любыми выражениями, которые можно оценить как истинные или ложные. Мы можем использовать операторы сравнения, которые изучили в предыдущей главе:

```
password = input("Введите пароль: ")
if len(password) < 8:
    print("Пароль слишком короткий")
elif password == "12345678":
    print("Этот пароль слишком простой")
elif password == password.lower():
    print("Добавьте заглавные буквы в пароль")
else:
    print("Пароль принят")
```

Этот пример демонстрирует, как можно использовать различные методы работы со строками в условиях. Мы проверяем длину пароля, сравниваем его с простым вариантом и проверяем, содержит ли он заглавные буквы.

3.2. Логические операторы: И, ИЛИ, НЕ

В реальной жизни наши решения редко основываются на одном-единственном факторе. Обычно мы учитываем несколько условий одновременно: «Если на улице солнечно И температура выше 20 градусов, то пойду гулять». Или: «Если сегодня выходной ИЛИ у меня отпуск, то буду спать до полудня».

Логические операторы (logical operators) позволяют комбинировать простые условия в более сложные логические выражения. В Python существует три основных логических оператора.

Оператор and (И) — возвращает `True` только тогда, когда оба условия истинны:

```
age = int(input("Введите ваш возраст: "))
has_license = input("Есть ли у вас водительские права? (да/нет): ")
if age >= 18 and has_license.lower() == "да":
    print("Вы можете арендовать автомобиль")
else:
    print("Извините, вы не можете арендовать автомобиль")
```

Оператор or (ИЛИ) — возвращает `True`, если истинно хотя бы одно из условий:

```
weather = input("Какая погода? (дождь/снег/солнце): ")
mood = input("Какое настроение? (плохое/хорошее): ")
if weather == "дождь" or mood == "плохое":
    print("Сегодня лучше остаться дома")
else:
    print("Отличный день для прогулки!")
```

Оператор not (НЕ) — инвертирует логическое значение, превращая `True` в `False` и наоборот:

```
is_weekend = input("Сегодня выходной? (да/нет): ")
if not is_weekend.lower() == "да":
    print("Пора на работу!")
else:
    print("Можно отдохнуть")
```

Логические операторы подчиняются определенным правилам приоритета. Оператор `not` имеет наивысший приоритет, затем идет `and`, и наконец `or`. Это означает, что выражение `A or B and not C` будет вычисляться как `A or (B and (not C))`.

Для ясности и предотвращения ошибок рекомендуется использовать скобки при создании сложных логических выражений:

```
temperature = float(input("Температура: "))
humidity = float(input("Влажность (%): "))
wind_speed = float(input("Скорость ветра (км/ч): "))
if (temperature > 20 and humidity < 60) or (temperature > 15 and wind_speed
< 10):
    print("Отличная погода для пикника!")
else:
    print("Возможно, стоит выбрать другой день")
```

Важно понимать, что Python использует так называемое «ленивое вычисление» (short-circuit evaluation) для логических операторов. Это означает, что если результат уже можно определить по первому условию, второе может не вычисляться:

```
age = int(input("Введите возраст: "))
# Если age >= 18 ложно, input() не выполнится
if age >= 18 and input("Подтвердите совершеннолетие (да/нет): ") == "да":
    print("Доступ разрешен")
```

Это поведение может быть полезным для предотвращения ошибок. Например, мы можем сначала проверить, что переменная не равна нулю, прежде чем делить на нее.

3.3. Сложные условия и их упрощение

По мере роста сложности программ наши условия могут становиться весьма громоздкими. Рассмотрим пример системы определения скидки в интернет-магазине:

```
age = int(input("Возраст покупателя: "))
is_student = input("Студент? (да/нет): ").lower() == "да"
is_premium = input("Премиум клиент? (да/нет): ").lower() == "да"
purchase_amount = float(input("Сумма покупки: "))
is_first_purchase = input("Первая покупка? (да/нет): ").lower() == "да"
if ((age < 25 and is_student) or (age > 65) or (is_premium and purchase_
amount > 1000) or
    (is_first_purchase and purchase_amount > 500)):
    discount = 0.15
    print(f"Поздравляем! Ваша скидка составляет {discount * 100}%")
else:
    discount = 0
    print("К сожалению, скидка не предоставляется")
```

Такие условия становятся трудночитаемыми и подверженными ошибкам. Существует несколько техник для их упрощения.

Техника 1. Использование промежуточных переменных

Сложное условие можно разбить на несколько логических частей, каждая из которых имеет осмысленное имя:

```
age = int(input("Возраст покупателя: "))
is_student = input("Студент? (да/нет): ").lower() == "да"
is_premium = input("Премиум клиент? (да/нет): ").lower() == "да"
purchase_amount = float(input("Сумма покупки: "))
is_first_purchase = input("Первая покупка? (да/нет): ").lower() == "да"
# Разбиваем сложное условие на понятные части
is_young_student = age < 25 and is_student
is_senior_citizen = age > 65
is_premium_big_purchase = is_premium and purchase_amount > 1000
is_first_big_purchase = is_first_purchase and purchase_amount > 500
# Теперь основное условие читается как естественное предложение
```

```
if is_young_student or is_senior_citizen or is_premium_big_purchase or is_
first_big_purchase:
    discount = 0.15
    print(f"Поздравляем! Ваша скидка составляет {discount * 100}%")
else:
    discount = 0
    print("К сожалению, скидка не предоставляется")
```

Техника 2. Использование вложенных условий для упрощения логики

Иногда проще использовать несколько простых условий вместо одного сложного:

```
username = input("Имя пользователя: ")
password = input("Пароль: ")
is_admin = input("Администратор? (да/нет): ").lower() == "да"
# Сначала проверяем базовые требования
if len(username) == 0 or len(password) == 0:
    print("Имя пользователя и пароль обязательны")
else:
    # Затем проверяем права доступа
    if is_admin:
        print("Добро пожаловать, администратор!")
        print("У вас есть полный доступ к системе")
    else:
        if len(password) >= 8:
            print("Добро пожаловать, пользователь!")
        else:
            print("Пароль должен содержать минимум 8 символов")
```

Техника 3. Принцип раннего возврата

Хотя мы пока не изучили функции, полезно знать о принципе, который можно применять с условиями. Вместо создания сложных вложенных структур можно последовательно проверять «стоп-условия»:

```
score = float(input("Введите результат теста (0-100): "))
# Проверяем некорректные данные в первую очередь
if score < 0 or score > 100:
    print("Некорректный результат теста")
elif score < 60:
    print("Тест не пройден. Попробуйте еще раз")
elif score < 80:
    print("Тест пройден удовлетворительно")
elif score < 95:
    print("Хороший результат!")
else:
    print("Отличный результат!")
```

Важно помнить о читаемости кода. Сложные условия должны быть понятными не только компьютеру, но и другим программистам (и вам самим через несколько ме-

сяцев). Хорошо написанное условие читается почти как обычное предложение на естественном языке.

При работе со сложными условиями полезно использовать законы логики для их упрощения. Например, закон Де Моргана гласит, что `not (A and B)` эквивалентно `(not A) or (not B)`, а `not (A or B)` эквивалентно `(not A) and (not B)`. Это может помочь в написании более понятных условий:

```
age = int(input("Возраст: "))
has_permission = input("Есть разрешение? (да/нет): ").lower() == "да"
# Вместо: if not (age < 18 or not has_permission):
if age >= 18 and has_permission:
    print("Доступ разрешен")
else:
    print("Доступ запрещен")
```

Помните, что цель упрощения условий — не только сделать код более читаемым, но и снизить вероятность ошибок. Простые, понятные условия легче тестировать и отлаживать, что особенно важно в сложных программных системах.

3.4. Вложенные условия: решения внутри решений

Жизнь редко предлагает нам простые бинарные выборы. Чаще всего, приняв одно решение, мы сталкиваемся с необходимостью принимать следующие решения, которые зависят от первого. Программирование отражает эту реальность через концепцию вложенных условий (nested conditions) — условий, размещенных внутри других условий.

Представьте процесс выбора одежды утром. Сначала вы смотрите на погоду — если солнечно, то выбираете между легкой и средней одеждой в зависимости от температуры. Если пасмурно, то решение снова зависит от температуры, но варианты уже другие. Каждое последующее решение принимается в контексте предыдущего.

```
weather = input("Какая погода? (солнечно/пасмурно/дождь): ").lower()
temperature = float(input("Температура воздуха: "))
if weather == "солнечно":
    if temperature > 25:
        print("Наденьте легкую одежду и солнцезащитные очки")
        print("Используйте солнцезащитный крем")
    elif temperature > 15:
        print("Подойдет легкая рубашка или футболка")
    else:
        print("Возьмите легкую куртку, несмотря на солнце")
elif weather == "пасмурно":
    if temperature > 20:
        print("Легкая одежда, но захватите легкую кофту")
    elif temperature > 10:
        print("Теплая одежда без зонта")
    else:
        print("Теплая одежда и возможно легкая куртка")
else:  # дождь
    print("Обязательно возьмите зонт или дождевик")
    if temperature < 15:
        print("И не забудьте про теплую одежду")
```

Вложенные условия создают древовидную структуру решений. Каждая ветвь дерева представляет собой определенный сценарий, и программа проходит по этому дереву, принимая решения на каждом уровне.

Рассмотрим более сложный пример — систему оценки кредитоспособности:

```
age = int(input("Возраст заявителя: "))
income = float(input("Месячный доход: "))
has_job = input("Есть ли постоянная работа? (да/нет): ").lower() == "да"
credit_history = input("Кредитная история (отличная/хорошая/плохая):
").lower()
if age >= 18:
    if has_job:
        if income >= 30000:
            if credit_history == "отличная":
                print("Кредит одобрен с льготной процентной ставкой")
                rate = 8.5
            elif credit_history == "хорошая":
                print("Кредит одобрен со стандартной процентной ставкой")
                rate = 12.0
            else:  # плохая кредитная история
                print("Кредит одобрен с повышенной процентной ставкой")
                rate = 18.0
        else:  # доход менее 30000
            if credit_history == "отличная" and income >= 20000:
                print("Кредит одобрен с ограниченной суммой")
                rate = 15.0
            else:
                print("К сожалению, кредит не может быть одобрен")
                rate = 0
    else:  # нет постоянной работы
        print("Для получения кредита необходима постоянная работа")
        rate = 0
else:  # возраст менее 18
    print("Кредиты выдаются только совершеннолетним")
    rate = 0
if rate > 0:
    print(f"Процентная ставка: {rate}%")
```

Хотя вложенные условия мощны, они могут быстро сделать код трудночитаемым. Существует негласное правило в программировании: избегать вложенности более трех уровней. Когда логика становится слишком сложной, стоит подумать о рефакторинге (refactoring).

Техника упрощения: инверсия условий

Вместо глубокой вложенности можно использовать «защитные условия» (guard clauses), которые проверяют исключительные случаи в первую очередь:

```
age = int(input("Возраст: "))
password = input("Пароль: ")
is_verified = input("Подтвержден email? (да/нет): ").lower() == "да"
# Сначала проверяем все "стоп-условия"
```

```
if age < 13:
    print("Регистрация доступна только с 13 лет")
elif len(password) < 8:
    print("Пароль должен содержать минимум 8 символов")
elif not is_verified:
    print("Необходимо подтвердить email-адрес")
else:
    # Основная логика выполняется, только если все проверки пройдены
    if age < 18:
        print("Регистрация с ограниченным функционалом")
        account_type = "junior"
    else:
        print("Полная регистрация завершена")
        account_type = "standard"

    print(f"Тип аккаунта: {account_type}")
```

Комбинирование вложенных условий с логическими операторами

Иногда можно заменить вложенность логическими операторами, но важно соблюдать баланс между краткостью и читаемостью:

```
user_type = input("Тип пользователя (admin/user/guest): ").lower()
time_of_day = int(input("Час (0-23): "))
# Менее читаемо, но более компактно
if (user_type == "admin") or (user_type == "user" and 9 <= time_of_day <=
18):
    print("Доступ к системе разрешен")
else:
    print("Доступ к системе запрещен")
# Более читаемо с вложенностью
if user_type == "admin":
    print("Доступ к системе разрешен (администратор)")
elif user_type == "user":
    if 9 <= time_of_day <= 18:
        print("Доступ к системе разрешен (рабочие часы)")
    else:
        print("Доступ к системе запрещен (нерабочие часы)")
else:
    print("Доступ к системе запрещен (недостаточно прав)")
```

3.5. Сопоставление с образцом (`match-case`)

В Python 3.10 появился новый мощный инструмент для работы с условиями — оператор `match-case`, также известный как сопоставление с образцом (pattern matching). Эта конструкция особенно полезна, когда нужно сравнить значение переменной с множеством возможных вариантов.

Традиционно подобные задачи решались с помощью длинных цепочек `if-elif-else`:

```
command = input("Введите команду (старт/стоп/пауза/сброс): ").lower()
if command == "старт":
    print("Система запускается...")
elif command == "стоп":
    print("Система останавливается...")
elif command == "пауза":
    print("Система приостанавливается...")
elif command == "сброс":
    print("Система перезапускается...")
else:
    print("Неизвестная команда")
```

С оператором `match-case` тот же код выглядит более элегантно и читаемо:

```
command = input("Введите команду (старт/стоп/пауза/сброс): ").lower()
match command:
    case "старт":
        print("Система запускается...")
    case "стоп":
        print("Система останавливается...")
    case "пауза":
        print("Система приостанавливается...")
    case "сброс":
        print("Система перезапускается...")
    case _:  # символ подчеркивания означает "любой другой случай"
        print("Неизвестная команда")
```

Оператор `match` принимает значение для сопоставления, а каждый case определяет возможный образец (pattern). Если значение совпадает с образцом, выполняется соответствующий блок кода. Символ `_` в последнем case играет роль `"catch-all"` — он срабатывает для всех значений, которые не подошли к предыдущим образцам.

Сопоставление с несколькими значениями

Один case может обрабатывать несколько значений одновременно, используя символ `|` (логическое ИЛИ):

```
day_of_week = input("Какой сегодня день недели?: ").lower()
match day_of_week:
    case "понедельник" | "вторник" | "среда" | "четверг" | "пятница":
        print("Рабочий день")
        print("Время для продуктивной работы!")
    case "суббота" | "воскресенье":
        print("Выходной день")
        print("Время для отдыха и хобби!")
    case _:
        print("Некорректный день недели")
```

Сопоставление с защитными условиями (`guard clauses`)

Можно добавлять дополнительные условия к образцам с помощью ключевого слова `if`:

```
age = int(input("Введите возраст: "))
match age:
    case n if n < 0:
        print("Возраст не может быть отрицательным")
    case n if 0 <= n < 13:
        print("Ребенок")
    case n if 13 <= n < 18:
        print("Подросток")
    case n if 18 <= n < 65:
        print("Взрослый")
    case n if n >= 65:
        print("Пенсионер")
```

Практический пример: калькулятор с `match-case`

Рассмотрим реализацию простого калькулятора:

```
print("Простой калькулятор")
print("Доступные операции: +, -, *, /, %, **")
num1 = float(input("Введите первое число: "))
operation = input("Введите операцию: ")
num2 = float(input("Введите второе число: "))
match operation:
    case "+":
        result = num1 + num2
        print(f"{num1} + {num2} = {result}")
    case "-":
        result = num1 - num2
        print(f"{num1} - {num2} = {result}")
    case "*":
        result = num1 * num2
        print(f"{num1} * {num2} = {result}")
    case "/":
        if num2 != 0:
            result = num1 / num2
            print(f"{num1} / {num2} = {result}")
        else:
            print("Ошибка: деление на ноль!")
    case "%":
        if num2 != 0:
            result = num1 % num2
            print(f"{num1} % {num2} = {result}")
        else:
            print("Ошибка: деление на ноль!")
    case "**":
        result = num1 ** num2
        print(f"{num1} ** {num2} = {result}")
    case _:
        print("Неподдерживаемая операция!")
```

Сопоставление со структурированными данными

Одной из мощных особенностей `match-case` является способность сопоставлять сложные структуры данных. Хотя мы пока не изучили списки и кортежи подробно, полезно увидеть возможности этого инструмента:

```
# Система обработки координат точки
user_input = input("Введите координаты через пробел (x y) или 'центр': ")
if user_input.lower() == "центр":
    coordinates = (0, 0)
else:
    coords = user_input.split()
    coordinates = (int(coords[0]), int(coords[1]))
match coordinates:
    case (0, 0):
        print("Точка находится в центре координат")
    case (0, y):
        print(f"Точка находится на оси Y в позиции {y}")
    case (x, 0):
        print(f"Точка находится на оси X в позиции {x}")
    case (x, y) if x > 0 and y > 0:
        print(f"Точка находится в первой четверти: ({x}, {y})")
    case (x, y) if x < 0 and y > 0:
        print(f"Точка находится во второй четверти: ({x}, {y})")
    case (x, y) if x < 0 and y < 0:
        print(f"Точка находится в третьей четверти: ({x}, {y})")
    case (x, y):
        print(f"Точка находится в четвертой четверти: ({x}, {y})")
```

Когда использовать `match-case`
Оператор `match-case` особенно полезен в следующих ситуациях.

1. **Множественный выбор по значению** — когда нужно сравнить переменную с несколькими константными значениями.
2. **Обработка команд** — в интерактивных программах и парсерах команд.
3. **Классификация данных** — когда нужно классифицировать входные данные по категориям.
4. **Конечные автоматы** — для реализации машин состояний.

Важно помнить, что `match-case` — это не просто синтаксический сахар для `if-elif-else`. Он предоставляет более мощные возможности сопоставления образцов и может сделать код более читаемым и менее подверженным ошибкам, особенно при работе со сложными структурами данных.

Однако для простых условий традиционные конструкции `if-elif-else` могут быть более понятными, особенно начинающим программистам. Выбор между ними должен основываться на конкретной задаче и требованиях к читаемости кода.

3.6. Практические примеры: калькулятор, игра в угадайку

Теория обретает смысл только в практическом применении. В этом разделе мы создадим два полноценных проекта, которые продемонстрируют все изученные кон-

цепции условных конструкций в действии. Эти программы не только закрепят полученные знания, но и покажут, как создавать интерактивные приложения, способные принимать решения и адаптироваться к действиям пользователя.

Проект 1. Интеллектуальный калькулятор

Создадим калькулятор, который не просто выполняет вычисления, но и анализирует результаты, предупреждает о возможных ошибках и предоставляет дополнительную информацию пользователю:

```
print("=== Интеллектуальный калькулятор ===")
print("Поддерживаемые операции:")
print("+ (сложение), - (вычитание), * (умножение)")
print("/ (деление), ** (возведение в степень), % (остаток от деления)")
print("sqrt (квадратный корень)")
print("-" * 40)
# Получаем первое число
first_number = float(input("Введите первое число: "))
# Получаем операцию
operation = input("Введите операцию: ").strip().lower()
# Для квадратного корня второе число не нужно
if operation == "sqrt":
    if first_number < 0:
        print("Ошибка: квадратный корень из отрицательного числа!")
    else:
        result = first_number ** 0.5
        print(f"Квадратный корень из {first_number} = {result}")

        # Дополнительная информация
        if result == int(result):
            print("Результат является целым числом!")
        if first_number == 1:
            print("Интересно: квадратный корень из 1 всегда равен 1")
else:
    # Для всех остальных операций нужно второе число
    second_number = float(input("Введите второе число: "))

    match operation:
        case "+":
            result = first_number + second_number
            print(f"{first_number} + {second_number} = {result}")

            # Анализ результата
            if result == 0:
                print("Интересно: сумма равна нулю!")
            elif result > 1000000:
                print("Получился очень большой результат!")

        case "-":
            result = first_number - second_number
            print(f"{first_number} - {second_number} = {result}")
```

```
            if result == 0:
                print("Числа оказались равными!")
            elif result < 0:
                print("Результат отрицательный")

        case "*":
            result = first_number * second_number
            print(f"{first_number} * {second_number} = {result}")

            # Особые случаи умножения
            if first_number == 0 or second_number == 0:
                print("Умножение на ноль всегда дает ноль")
            elif first_number == 1:
                print("Умножение на 1 не изменяет число")
            elif second_number == 1:
                print("Умножение на 1 не изменяет число")
            elif result < 0:
                 print("При умножении чисел с разными знаками результат от-
рицательный")

        case "/":
            if second_number == 0:
                print("Ошибка: деление на ноль невозможно!")
            else:
                result = first_number / second_number
                print(f"{first_number} / {second_number} = {result}")

                # Анализ деления
                if result == 1:
                    print("Числа равны!")
                elif result == int(result):
                    print("Результат деления — целое число")
                     print(f"Это означает, что {first_number} нацело делит-
ся на {second_number}")
                else:
                     print(f"Результат: {result:.6f} (показано шесть знаков
после запятой)")

        case "**":
            # Проверяем особые случаи возведения в степень
            if first_number == 0 and second_number <= 0:
               print("Ошибка: 0 нельзя возводить в отрицательную степень!")
            else:
                result = first_number ** second_number
                print(f"{first_number} ** {second_number} = {result}")

                # Анализ результата
                if second_number == 0:
                    print("Любое число в степени 0 равно 1 (кроме некоторых
особых случаев)")
```

```
                elif second_number == 1:
                    print("Любое число в первой степени равно самому себе")
                elif second_number == 2:
                    print("Это возведение в квадрат")
                elif second_number == 3:
                    print("Это возведение в куб")

        case "%":
            if second_number == 0:
                print("Ошибка: деление на ноль невозможно!")
            else:
                result = first_number % second_number
                print(f"{first_number} % {second_number} = {result}")

                if result == 0:
                 print(f"{first_number} нацело делится на {second_number}")
                else:
                     print(f"При делении {first_number} на {second_number}
остаток составляет {result}")

        case _:
            print("Ошибка: неподдерживаемая операция!")
            print("Доступные операции: +, -, *, /, **, %, sqrt")
print("\nСпасибо за использование калькулятора!")
```

Проект 2. Игра «Угадай число» с адаптивным интеллектом

Создадим игру, которая не только проверяет догадки игрока, но и анализирует его стратегию, дает подсказки и адаптируется к уровню игрока:

```
import random
print("=== ИГРА 'УГАДАЙ ЧИСЛО' ===")
print("Я загадал число от 1 до 100.")
print("Ваша задача – угадать его за минимальное количество попыток!")
print("Я буду давать подсказки: 'больше', 'меньше' или 'угадал'")
print("-" * 50)
# Выбираем уровень сложности
difficulty = input("Выберите уровень сложности (легкий/средний/сложный):
").lower()
match difficulty:
    case "легкий":
        max_attempts = 10
        min_number = 1
        max_number = 50
       print(f"Легкий уровень: число от {min_number} до {max_number}, {max_
attempts} попыток")
    case "средний":
        max_attempts = 7
        min_number = 1
        max_number = 100
```

```
        print(f"Средний уровень: число от {min_number} до {max_number}, {max_
attempts} попыток")
    case "сложный":
        max_attempts = 5
        min_number = 1
        max_number = 200
        print(f"Сложный уровень: число от {min_number} до {max_number}, {max_
attempts} попыток")
    case _:
        print("Неизвестный уровень, устанавливаю средний уровень")
        max_attempts = 7
        min_number = 1
        max_number = 100
# Загадываем число
secret_number = random.randint(min_number, max_number)
# Переменные для анализа игры
attempts_used = 0
previous_guess = None
getting_closer = True
last_distance = max_number
print(f"\nЧисло загадано! У вас есть {max_attempts} попыток.")
print("Начинаем игру!\n")
# Основной игровой цикл
while attempts_used < max_attempts:
    # Получаем догадку от игрока
    try:
        current_guess = int(input(f"Попытка {attempts_used + 1}: Ваше чис-
ло: "))
    except ValueError:
        print("Пожалуйста, введите целое число!")
        continue

    # Проверяем корректность ввода
    if current_guess < min_number or current_guess > max_number:
        print(f"Число должно быть от {min_number} до {max_number}!")
        continue

    attempts_used += 1

    # Анализируем догадку
    if current_guess == secret_number:
        print(f" ПОЗДРАВЛЯЮ! Вы угадали число {secret_number}!")

        # Анализ результата
        if attempts_used == 1:
            print("Невероятно! Угадали с первой попытки!")
        elif attempts_used <= max_attempts // 2:
            print("Отличный результат! Вы очень быстро справились!")
        elif attempts_used < max_attempts:
            print("Хороший результат! Уложились в лимит попыток!")
```

```
            print(f"Использовано попыток: {attempts_used} из {max_attempts}")
            break

        else:
            # Анализируем направление поиска
            current_distance = abs(current_guess - secret_number)

            if current_guess < secret_number:
                print("Загаданное число БОЛЬШЕ")
                hint = "больше"
            else:
                print("Загаданное число МЕНЬШЕ")
                hint = "меньше"

            # Даем дополнительные подсказки
            if current_distance <= 5:
                print("Очень горячо! Совсем близко!")
            elif current_distance <= 10:
                print("Горячо! Вы близко к цели!")
            elif current_distance <= 20:
                print("Тепло, но можно ближе")
            else:
                print("Холодно, вы далеко от цели")

            # Анализируем стратегию игрока
            if previous_guess is not None:
                if current_distance < last_distance:
                    print("Хорошо! Вы приближаетесь к ответу!")
                    getting_closer = True
                elif current_distance > last_distance:
                    print("Вы отдаляетесь от правильного ответа")
                    getting_closer = False
                else:
                    print("Расстояние до цели не изменилось")

            # Стратегические советы
            if attempts_used == max_attempts // 2:
                if getting_closer:
                    print("Совет: Продолжайте двигаться в том же направлении!")
                else:
                    print("Совет: Попробуйте поискать в другом диапазоне")

            # Предупреждение о заканчивающихся попытках
            remaining = max_attempts - attempts_used
            if remaining == 1:
                print("ВНИМАНИЕ! Остается всего 1 попытка!")
            elif remaining == 2:
                print("Осталось только 2 попытки!")

            last_distance = current_distance
            previous_guess = current_guess
```

```
# Проверяем, закончились ли попытки
if attempts_used >= max_attempts and current_guess != secret_number:
    print(f"\n Игра окончена! Попытки закончились.")
    print(f"Загаданное число было: {secret_number}")

    # Анализ неудачной попытки
    final_distance = abs(current_guess - secret_number)
    if final_distance <= 10:
        print("Вы были очень близко! Попробуйте еще раз!")
    elif final_distance <= 25:
        print("Неплохая попытка, но можно лучше!")
    else:
        print("В следующий раз попробуйте более систематичный подход!")
# Предложение повторить игру
play_again = input("\nХотите сыграть еще раз? (да/нет): ").lower()
if play_again == "да" or play_again == "yes" or play_again == "y":
    print("Отлично! Перезапустите программу для новой игры!")
else:
    print("Спасибо за игру! До свидания!")
```

Эти примеры демонстрируют, как условные конструкции превращают простые программы в интеллектуальные системы, способные анализировать ситуацию, принимать решения и взаимодействовать с пользователем осмысленным образом.

3.7. Задачи для самостоятельного решения

Практика — основа мастерства в программировании. Следующие задачи организованы по возрастанию сложности и помогут вам закрепить все изученные концепции условных конструкций. Рекомендуется решать задачи последовательно, не пропуская более простые варианты.

Задача 1. Определение времени суток

Напишите программу, которая принимает час (число от 0 до 23) и определяет время суток:

- 6–11: утро
- 12–17: день
- 18–22: вечер
- 23–5: ночь

Добавьте проверку корректности введенного часа.

Задача 2. Классификация треугольников

Программа получает три стороны треугольника и определяет его тип:

- Равносторонний (все стороны равны)
- Равнобедренный (две стороны равны)
- Разносторонний (все стороны разные)
- Некорректный (не может существовать)

Помните: треугольник существует, если сумма любых двух сторон больше третьей.

Задача 3. Проверка високосного года
Создайте программу для определения високосного года. Год является високосным, если:

- Он делится на 4 И не делится на 100
- ИЛИ он делится на 400

Задача 4. Система скидок в магазине
Разработайте систему расчета скидки со следующими правилами:

- Если сумма покупки > 5000 руб. — скидка 10%
- Если покупатель пенсионер (возраст ≥ 65) — дополнительно 5%
- Если сегодня день рождения покупателя — дополнительно 5%
- Если клиент VIP — скидка 15% (не суммируется с обычной, берется максимальная)
- Максимальная скидка не может превышать 25%

Задача 5. Анализатор пароля
Создайте программу для проверки надежности пароля:

- Длина минимум восемь символов
- Содержит заглавные буквы
- Содержит строчные буквы
- Содержит цифры
- Не содержит пробелов
- Не является одним из слабых паролей: "password", "12345678", "qwerty"

Выводите уровень надежности: слабый, средний, сильный.

Задача 6. Определение сезона и рекомендации
Программа получает месяц и температуру, затем:

- Определяет сезон года
- Дает рекомендации по одежде с учетом сезона и температуры
- Предупреждает об экстремальных условиях (жара > 35°C, холод < -20°C)

Задача 7. Конвертер единиц измерения
Используя `match-case`, создайте универсальный конвертер:

- Длина: мм, см, м, км, дюймы, футы
- Вес: г, кг, фунты
- Температура: Цельсий, Фаренгейт, Кельвин

Программа должна запрашивать тип измерения, исходную единицу, целевую единицу и значение.

Задача 8. Игра «Камень, ножницы, бумага»
Реализуйте игру против компьютера с использованием `match-case`:

- Компьютер делает случайный выбор
- Определяется победитель согласно правилам
- Ведется счет игр

- После каждого раунда предлагается продолжить или завершить игру
- В конце выводится статистика

Задача 9. Система оценки риска кредита
Создайте экспертную систему для оценки кредитного риска:
Входные параметры:

- Возраст заемщика
- Ежемесячный доход
- Размер запрашиваемого кредита
- Срок кредита (месяцы)
- Кредитная история (отличная/хорошая/удовлетворительная/плохая)
- Наличие поручителей
- Наличие залога

Система должна:

- Рассчитать соотношение кредита к доходу
- Определить категорию риска (низкий/средний/высокий/критический)
- Предложить процентную ставку
- Дать рекомендацию по одобрению/отклонению

Задача 10. Персональный помощник погоды
Создайте интеллектуального помощника, который:

- Получает данные о погоде (температура, влажность, ветер, осадки)
- Учитывает планы пользователя (работа, спорт, прогулка, путешествие)
- Дает персонализированные рекомендации по одежде
- Предупреждает о неблагоприятных условиях
- Предлагает альтернативы планов при плохой погоде

Каждая задача — это мини-проект, который поможет вам понять, как условные конструкции применяются в реальных программах. Не стремитесь решить все задачи за один присест — лучше качественно проработать несколько, чем поверхностно подойти ко многим.

Помните: в программировании нет единственно правильного решения. Ваш код может отличаться от решений других людей, и это нормально. Главное — чтобы программа работала корректно, была понятной и решала поставленную задачу.

Глава 4. Циклы — автоматизация повторений

4.1. Философия циклов: зачем повторять код

Представьте, что вам необходимо вывести числа от 1 до 100. Имея знания, полученные в предыдущих главах, вы могли бы написать сто команд `print()`, но даже самый терпеливый программист понимает абсурдность такого подхода. Именно здесь в игру вступают циклы (loops) — одна из фундаментальных конструкций программирования, позволяющая автоматизировать повторяющиеся действия.

Циклы решают проблему дублирования кода, следуя важнейшему принципу программирования DRY (Don't Repeat Yourself — «Не повторяй себя»). Когда мы видим, что определенная последовательность действий должна выполняться многократно, цикл становится естественным решением. Это не просто сокращение объема кода — это принципиально иной способ мышления о задачах.

В реальном мире циклы повсеместны. Банковская система каждую ночь обрабатывает миллионы транзакций, веб-сервер отвечает на тысячи запросов в секунду, поисковая система анализирует содержимое веб-страниц. Все эти процессы основаны на циклах — повторении определенных операций до достижения цели или исчерпания данных.

Python предоставляет два основных типа циклов: `for` для случаев, когда количество итераций известно заранее или определяется размером коллекции данных, и `while` для ситуаций, когда повторение продолжается до выполнения определенного условия. Понимание различий между этими конструкциями и умение выбрать подходящую — важная часть мастерства программиста.

Циклы также открывают путь к более сложным алгоритмическим концепциям. Поиск элемента в списке, вычисление факториала, обработка файлов — все эти задачи требуют циклических конструкций. Более того, понимание циклов подготавливает основу для изучения рекурсии, функций высшего порядка и других продвинутых техник программирования.

4.2. Цикл `for`: известное количество итераций

Цикл `for` в Python — это элегантная конструкция для итерации по последовательностям данных. В отличие от многих других языков программирования, где `for` работает с числовыми счетчиками, Python использует концепцию итерации по объектам, что делает код более читаемым и выразительным.

Базовый синтаксис цикла `for` выглядит следующим образом:

```
for переменная in последовательность:
    # блок кода, выполняемый на каждой итерации
    действие()
```

Ключевое слово `for` открывает цикл, за ним следует переменная-итератор, которая будет принимать значения из последовательности на каждом шаге. Слово `in` указы-

вает принадлежность, а двоеточие завершает заголовок цикла. Все действия, которые должны повторяться, записываются с отступом в четыре пробела.

Рассмотрим простейший пример — итерацию по строке:

```
message = "Python"
for letter in message:
    print(f"Буква: {letter}")
```

Этот код выведет каждую букву слова "Python" на отдельной строке. Переменная `letter` последовательно принимает значения `'P'`, `'y'`, `'t'`, `'h'`, `'o'`, `'n'`, и для каждого значения выполняется команда вывода.

Для работы с числовыми последовательностями Python предоставляет функцию `range()`, которая генерирует арифметические прогрессии. Функция `range()` может принимать от одного до трех аргументов:

```
# range(stop) - от 0 до stop-1
for i in range(5):
    print(i)  # Выведет: 0, 1, 2, 3, 4
# range(start, stop) - от start до stop-1
for i in range(2, 7):
    print(i)  # Выведет: 2, 3, 4, 5, 6
# range(start, stop, step) - с определенным шагом
for i in range(0, 10, 2):
    print(i)  # Выведет: 0, 2, 4, 6, 8
```

Важно понимать, что `range()` не создает список в памяти — это специальный объект-генератор, который вычисляет значения по требованию. Это обеспечивает эффективность при работе с большими диапазонами чисел.

Практический пример — вычисление суммы квадратов чисел от 1 до n:

```
n = 10
sum_of_squares = 0
for i in range(1, n + 1):
    sum_of_squares += i ** 2
    print(f"i = {i}, i² = {i**2}, сумма = {sum_of_squares}")
print(f"Сумма квадратов чисел от 1 до {n}: {sum_of_squares}")
```

Этот код демонстрирует накопление результата в переменной `sum_of_squares`, что является типичным паттерном при работе с циклами. Каждая итерация добавляет квадрат текущего числа к общей сумме.

Цикл `for` может работать с любыми итерируемыми объектами (iterables). Строки, как мы уже видели, являются итерируемыми, но в будущем мы изучим списки, кортежи и словари, которые также поддерживают итерацию.

Рассмотрим более сложный пример — анализ текста:

```
text = "Hello World"
vowels = "aeiouAEIOU"
vowel_count = 0
consonant_count = 0
for char in text:
    if char.isalpha():  # Проверяем, является ли символ буквой
        if char in vowels:
            vowel_count += 1
```

```
            print(f"'{char}' - гласная")
        else:
            consonant_count += 1
            print(f"'{char}' - согласная")
print(f"Гласных: {vowel_count}, согласных: {consonant_count}")
```

Этот пример показывает, как циклы позволяют анализировать данные посимвольно, применяя различные условия и накапливая статистику. Метод `isalpha()` проверяет, является ли символ буквой, а операция `in` определяет принадлежность символа к множеству гласных.

4.3. Цикл `while`: до тех пор, пока...

Если цикл `for` предназначен для итерации по известным последовательностям, то цикл `while` создан для ситуаций, когда количество повторений заранее неизвестно. Он продолжает выполнение до тех пор, пока определенное условие остается истинным, что делает его незаменимым для решения многих алгоритмических задач.

Синтаксис цикла `while` напоминает условную конструкцию `if`:

```
while условие:
    # блок кода, выполняемый, пока условие истинно
    действие()
    # важно: где-то внутри цикла условие должно измениться
```

Критически важно понимать, что внутри цикла `while` должны происходить действия, которые в конечном итоге сделают условие ложным. В противном случае возникнет бесконечный цикл (infinite loop), который будет выполняться до принудительного завершения программы.

Простейший пример — обратный отсчет:

```
count = 5
while count > 0:
    print(f"До запуска: {count}")
    count -= 1  # Уменьшаем счетчик
print("Запуск!")
```

В этом примере переменная `count` играет роль счетчика цикла. На каждой итерации ее значение уменьшается на единицу, и когда оно достигает нуля, условие `count > 0` становится ложным, и цикл завершается.

Цикл `while` особенно удобен для валидации пользовательского ввода:

```
while True:  # Бесконечный цикл
    user_input = input("Введите положительное число (или 'exit' для выхода): ")

    if user_input.lower() == 'exit':
        print("Программа завершена.")
        break  # Выход из цикла

    try:
        number = float(user_input)
        if number > 0:
```

```
            print(f"Отлично! Вы ввели: {number}")
            break  # Выход из цикла после корректного ввода
        else:
            print("Число должно быть положительным. Попробуйте снова.")
    except ValueError:
        print("Это не число. Попробуйте снова.")
```

Этот пример демонстрирует несколько важных концепций. Использование `while True` создает бесконечный цикл, который завершается только при выполнении команды `break`. Конструкция `try-except` обрабатывает возможные ошибки при преобразовании строки в число, что является хорошей практикой программирования.

Рассмотрим классический алгоритм — вычисление наибольшего общего делителя (НОД) двух чисел методом Евклида:

```
def gcd(a, b):
    print(f"Вычисляем НОД({a}, {b})")

    while b != 0:
        print(f"a = {a}, b = {b}")
        remainder = a % b  # Остаток от деления
        a = b
        b = remainder
        print(f"Новые значения: a = {a}, b = {b}")

    return a
# Пример использования
number1 = 48
number2 = 18
result = gcd(number1, number2)
print(f"НОД({number1}, {number2}) = {result}")
```

Алгоритм Евклида элегантно демонстрирует мощь цикла `while`. Мы не знаем заранее, сколько итераций потребуется для нахождения НОД, но знаем, что процесс должен продолжаться до тех пор, пока остаток от деления не станет равным нулю.

Важно понимать различия в применении циклов `for` и `while`. Используйте `for`, когда:

- нужно обработать каждый элемент коллекции;
- количество итераций определяется размером данных;
- требуется выполнить действие заданное количество раз.

Используйте `while`, когда:

- условие завершения основано на логическом выражении;
- количество итераций заранее неизвестно;
- нужно продолжать выполнение до достижения определённого состояния.

Рассмотрим практический пример — игру «Угадай число»:

```
import random
secret_number = random.randint(1, 100)
attempts = 0
```

```
max_attempts = 7
print("Добро пожаловать в игру 'Угадай число'!")
print(f"Я загадал число от 1 до 100. У вас есть {max_attempts} попыток.")
while attempts < max_attempts:
    try:
        guess = int(input(f"Попытка {attempts + 1}: Введите ваше число: "))
        attempts += 1

        if guess == secret_number:
          print(f"Поздравляю! Вы угадали число {secret_number} за {attempts}
попыток!")
            break
        elif guess < secret_number:
            print("Загаданное число больше.")
        else:
            print("Загаданное число меньше.")

        remaining = max_attempts - attempts
        if remaining > 0:
            print(f"Осталось попыток: {remaining}")

    except ValueError:
        print("Пожалуйста, введите целое число.")
        # attempts не увеличиваем, так как это была некорректная попытка
# Если цикл завершился без break
if attempts >= max_attempts and guess != secret_number:
    print(f"Попытки исчерпаны! Загаданное число было: {secret_number}")
```

Этот пример объединяет множество концепций: цикл `while` с ограничением по количеству попыток, обработку исключений, условную логику и взаимодействие с пользователем. Программа демонстрирует, как циклы позволяют создавать интерактивные приложения с динамическим поведением.

4.4. Управление циклами: `break` и `continue`

В процессе выполнения циклов часто возникают ситуации, когда необходимо изменить естественный ход выполнения. Возможно, нужно досрочно прервать цикл при достижении определенного условия или пропустить текущую итерацию и перейти к следующей. Python предоставляет две специальные команды для управления циклами: `break` и `continue`.

Команда `break`: досрочный выход

Команда `break` немедленно завершает выполнение цикла, передавая управление первой команде после цикла. Это особенно полезно в ситуациях, когда продолжение цикла становится бессмысленным или когда найден искомый результат.

```
# Поиск первого четного числа в последовательности
numbers = [1, 3, 7, 8, 9, 12, 15]
for number in numbers:
```

```
    print(f"Проверяем число: {number}")
    if number % 2 == 0:
        print(f"Найдено первое четное число: {number}")
        break
    print("Число нечетное, продолжаем поиск")
print("Поиск завершен")
```

В этом примере цикл прекращается, как только встречается первое четное число. Без `break` пришлось бы проверять все элементы списка, даже после нахождения результата.

Команда `break` особенно важна в циклах `while`, где может потребоваться выход по нескольким различным условиям:

```
balance = 1000
transaction_fee = 50
while True:
    print(f"Текущий баланс: {balance} рублей")
      action = input("Выберите действие (withdraw/deposit/balance/exit):
").lower()

    if action == 'exit':
        print("Спасибо за использование банкомата!")
        break
    elif action == 'balance':
        continue  # Баланс уже выведен, переходим к следующей итерации
    elif action == 'withdraw':
        try:
            amount = float(input("Сумма для снятия: "))
            total_cost = amount + transaction_fee

            if total_cost > balance:
                print("Недостаточно средств с учетом комиссии!")
                continue

            balance -= total_cost
            print(f"Снято: {amount}, комиссия: {transaction_fee}")

        except ValueError:
            print("Некорректная сумма")
    elif action == 'deposit':
        try:
            amount = float(input("Сумма для пополнения: "))
            if amount <= 0:
                print("Сумма должна быть положительной")
                continue
            balance += amount
            print(f"Счёт пополнен на {amount} рублей")
        except ValueError:
            print("Некорректная сумма")
    else:
        print("Неизвестная команда")
```

Команда continue: пропуск итерации

Команда continue пропускает оставшуюся часть текущей итерации и переходит к следующему циклу. В цикле for это означает переход к следующему элементу последовательности, а в цикле while — к проверке условия продолжения цикла.

```
# Обработка только положительных чисел
numbers = [-2, 5, -1, 0, 8, -3, 12]
sum_positive = 0
for number in numbers:
    if number <= 0:
        print(f"Пропускаем неположительное число: {number}")
        continue

    print(f"Добавляем к сумме: {number}")
    sum_positive += number
print(f"Сумма положительных чисел: {sum_positive}")
```

Команда continue помогает избежать глубокой вложенности условий, делая код более читаемым. Сравните два варианта:

```
# Без continue - глубокая вложенность
for i in range(1, 21):
    if i % 2 == 0:  # Только четные числа
        if i % 4 != 0:  # Исключаем числа, кратные 4
            if i < 15:  # Только числа меньше 15
                print(f"Обрабатываем: {i}")
# С continue - плоская структура
for i in range(1, 21):
    if i % 2 != 0:  # Пропускаем нечетные
        continue
    if i % 4 == 0:  # Пропускаем кратные 4
        continue
    if i >= 15:  # Пропускаем большие числа
        continue

    print(f"Обрабатываем: {i}")
```

Второй вариант с continue читается как серия фильтров, что делает логику более понятной.

Практический пример: валидация данных

```
# Проверка корректности email-адресов
email_list = [
    "user@example.com",
    "invalid.email",
    "test@domain.ru",
    "",
    "another@site.org",
    "bad@",
    "good@company.com"
```

```
]
valid_emails = []
invalid_count = 0
for email in email_list:
    # Проверяем на пустую строку
    if not email:
        print("Пропускаем пустую строку")
        invalid_count += 1
        continue

    # Простейшая проверка на наличие @ и точки
    if '@' not in email:
        print(f"Email без @: {email}")
        invalid_count += 1
        continue

    # Разделяем на локальную и доменную части
    parts = email.split('@')
    if len(parts) != 2:
        print(f"Некорректный формат: {email}")
        invalid_count += 1
        continue

    local_part, domain = parts

    # Проверяем локальную часть
    if not local_part:
        print(f"Пустая локальная часть: {email}")
        invalid_count += 1
        continue

    # Проверяем доменную часть
    if '.' not in domain or not domain:
        print(f"Некорректный домен: {email}")
        invalid_count += 1
        continue

    # Если дошли до этого места — email прошел все проверки
    print(f"Корректный email: {email}")
    valid_emails.append(email)
print(f"\nРезультат: {len(valid_emails)} корректных, {invalid_count} некор-
ректных")
print("Корректные email-адреса:", valid_emails)
```

4.5. Вложенные циклы: циклы внутри циклов

Когда данные имеют многомерную структуру или задача требует работы с комбинациями элементов, возникает необходимость в использовании вложенных циклов (nested loops) — конструкций, где один цикл выполняется внутри другого. Это мощ-

ный инструмент, но он требует осторожного подхода к пониманию потока выполнения и оценке вычислительной сложности.

Принцип работы вложенных циклов

Во вложенных циклах внешний цикл управляет общим количеством групп итераций, а внутренний цикл полностью выполняется для каждого значения внешнего цикла. Если внешний цикл имеет `n` итераций, а внутренний — `m`, то общее количество операций составит `n × m`.

```
# Таблица умножения 5×5
print("Таблица умножения:")
print("    ", end="")
# Выводим заголовок
for j in range(1, 6):
    print(f"{j:4}", end="")
print()  # Переход на новую строку
# Выводим строки таблицы
for i in range(1, 6):
    print(f"{i:2}:", end="")  # Номер строки
    for j in range(1, 6):
        result = i * j
        print(f"{result:4}", end="")
    print()  # Переход на новую строку после каждой строки
```

Этот пример создает таблицу умножения, где внешний цикл отвечает за строки, а внутренний — за столбцы. Каждая итерация внешнего цикла запускает полный цикл внутреннего.

Работа с двумерными структурами

Вложенные циклы естественно подходят для работы с матрицами и таблицами данных:

```
# Создание и обработка матрицы
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
print("Исходная матрица:")
for row in matrix:
    for element in row:
        print(f"{element:3}", end=" ")
    print()  # Новая строка после каждой строки матрицы
# Поиск максимального элемента
max_value = matrix[0][0]  # Предполагаем, что первый элемент максимальный
max_row, max_col = 0, 0
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        if matrix[i][j] > max_value:
            max_value = matrix[i][j]
```

```
            max_row, max_col = i, j
print(f"\nМаксимальный элемент: {max_value}")
print(f"Позиция: строка {max_row + 1}, столбец {max_col + 1}")
# Сумма элементов по строкам
print("\nСуммы по строкам:")
for i in range(len(matrix)):
    row_sum = 0
    for j in range(len(matrix[i])):
        row_sum += matrix[i][j]
    print(f"Строка {i + 1}: {row_sum}")
```

Генерация комбинаций

Вложенные циклы незаменимы для генерации всех возможных комбинаций элементов:

```
# Генерация всех возможных паролей из двух символов
# используя цифры и строчные буквы
digits = "0123456789"
letters = "abcdefghijklmnopqrstuvwxyz"
password_count = 0
sample_passwords = []
print("Примеры паролей из двух символов:")
for first_char in digits + letters:
    for second_char in digits + letters:
        password = first_char + second_char
        password_count += 1

        # Сохраняем только первые 10 для демонстрации
        if len(sample_passwords) < 10:
            sample_passwords.append(password)
print("Первые 10 паролей:", sample_passwords)
print(f"Всего возможных паролей: {password_count}")
# Расчёт: (10 цифр + 26 букв) × (10 цифр + 26 букв) = 36 × 36 = 1296
```

Управление выполнением вложенных циклов

Команды `break` и `continue` во вложенных циклах действуют только на текущий уровень:

```
# Поиск первого совпадения в матрице
target = 7
matrix = [
    [1, 3, 5],
    [2, 7, 9],
    [4, 6, 8]
]
found = False
for i in range(len(matrix)):
    print(f"Проверяем строку {i + 1}")
```

```
    for j in range(len(matrix[i])):
        print(f"  Проверяем элемент {matrix[i][j]}")

        if matrix[i][j] == target:
            print(f"Найдено! Элемент {target} в позиции ({i}, {j})")
            found = True
            break  # Выходим из внутреннего цикла

    if found:
        break  # Выходим из внешнего цикла
if not found:
    print(f"Элемент {target} не найден")
```

Оптимизация вложенных циклов

При работе с вложенными циклами важно учитывать вычислительную сложность. Рассмотрим задачу поиска дубликатов в списке:

```
# Неэффективный способ - O(n²)
numbers = [3, 7, 2, 7, 1, 3, 9, 2]
duplicates = []
print("Поиск дубликатов (неэффективный способ):")
comparisons = 0
for i in range(len(numbers)):
    for j in range(i + 1, len(numbers)):  # Начинаем с i+1, чтобы не сравнивать элемент с самим собой
        comparisons += 1
        if numbers[i] == numbers[j]:
            if numbers[i] not in duplicates:
                duplicates.append(numbers[i])
                print(f"Найден дубликат: {numbers[i]} (позиции {i} и {j})")
print(f"Количество сравнений: {comparisons}")
print(f"Дубликаты: {duplicates}")
# Более эффективный способ с использованием подсчета
print("\nПоиск дубликатов (эффективный способ):")
number_counts = {}
# Подсчитываем вхождения каждого числа
for number in numbers:
    if number in number_counts:
        number_counts[number] += 1
    else:
        number_counts[number] = 1
# Находим числа, встречающиеся более одного раза
efficient_duplicates = []
for number, count in number_counts.items():
    if count > 1:
        efficient_duplicates.append(number)
print(f"Дубликаты: {efficient_duplicates}")
print(f"Количество операций: {len(numbers)} (линейная сложность)")
```

4.6. Генераторы списков: элегантность Python

Генераторы списков (list comprehensions) — это одна из наиболее элегантных и мощных особенностей Python, позволяющая создавать новые списки на основе существующих данных в краткой и выразительной форме. Эта конструкция объединяет цикл, условную логику и создание списка в одном выражении, часто заменяя несколько строк традиционного кода одной строкой.

Базовый синтаксис генераторов списков

Основная форма генератора списков выглядит следующим образом:

```
новый_список = [выражение for элемент in итерируемый_объект]
Это эквивалентно традиционному циклу:
новый_список = []
for элемент in итерируемый_объект:
    новый_список.append(выражение)
```

Рассмотрим простой пример — создание списка квадратов чисел:

```
# Традиционный способ
squares_traditional = []
for x in range(1, 11):
    squares_traditional.append(x ** 2)
# Генератор списков
squares_comprehension = [x ** 2 for x in range(1, 11)]
print("Традиционный способ:", squares_traditional)
print("Генератор списков:", squares_comprehension)
print("Результаты идентичны:", squares_traditional == squares_comprehension)
```

Генераторы с условиями

Генераторы списков могут включать условие для фильтрации элементов:

```
новый_список = [выражение for элемент in итерируемый_объект if условие]
```

Практические примеры:

```
# Только четные квадраты
even_squares = [x ** 2 for x in range(1, 21) if x % 2 == 0]
print("Квадраты четных чисел от 1 до 20:", even_squares)
# Фильтрация и преобразование строк
words = ["Python", "programming", "is", "amazing", "and", "powerful"]
long_words_upper = [word.upper() for word in words if len(word) > 5]
print("Длинные слова заглавными буквами:", long_words_upper)
# Извлечение чисел из смешанного списка
mixed_data = [1, "hello", 3.14, "world", 42, True, "Python", 7.5]
numbers_only = [item for item in mixed_data if isinstance(item, (int, float))
and not isinstance(item, bool)]
print("Только числа:", numbers_only)
```

Сложные выражения в генераторах

Генераторы списков могут содержать сложные выражения и вызовы функций:

```
# Обработка текстовых данных
sentences = [
    "Python is a programming language",
    "List comprehensions are powerful",
    "Code readability counts"
]
# Подсчет слов в каждом предложении
word_counts = [len(sentence.split()) for sentence in sentences]
print("Количество слов в предложениях:", word_counts)
# Создание словаря из списка (возвращаем список кортежей)
sentence_stats = [(sentence, len(sentence.split())) for sentence in
sentences]
print("Статистика предложений:")
for sentence, word_count in sentence_stats:
    print(f"'{sentence[:30]}...': {word_count} слов")
# Математические операции
import math
angles = [0, 30, 45, 60, 90]
# Преобразуем градусы в радианы и вычисляем синус
sin_values = [round(math.sin(math.radians(angle)), 3) for angle in angles]
print("Углы (градусы):", angles)
print("Синусы углов:", sin_values)
```

Вложенные генераторы списков

Для работы с двумерными данными можно использовать вложенные генераторы:

```
# Создание таблицы умножения как список списков
multiplication_table = [[i * j for j in range(1, 6)] for i in range(1, 6)]
print("Таблица умножения 5×5:")
for row in multiplication_table:
    print([f"{num:2}" for num in row])
# Извлечение всех элементов из двумерного списка в одномерный
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [element for row in matrix for element in row]
print("Исходная матрица:", matrix)
print("Преобразованная в одномерный список:", flattened)
# Поиск позиций элементов, удовлетворяющих условию
coordinates = [(i, j) for i in range(3) for j in range(3) if matrix[i][j]
% 2 == 0]
print("Координаты четных элементов:", coordinates)
```

Условные выражения в генераторах

Генераторы списков поддерживают тернарный оператор для создания условных выражений:

```
# Классификация чисел как четных или нечетных
```

```
numbers = range(1, 11)
parity = [f"{num} is {'even' if num % 2 == 0 else 'odd'}" for num in numbers]
print("Классификация чисел:")
for classification in parity:
    print(classification)
# Нормализация температур
temperatures_celsius = [-10, 0, 15, 25, 35, 40]
temperature_descriptions = [
    f"{temp}°C ({'freezing' if temp < 0 else 'cold' if temp < 10 else 'mild'
if temp < 25 else 'warm' if temp < 35 else 'hot'})"
    for temp in temperatures_celsius
]
print("Описание температур:")
for desc in temperature_descriptions:
    print(desc)
```

Производительность и читаемость

Генераторы списков не только делают код более кратким, но часто работают быстрее эквивалентных циклов:

```
import time
# Большой диапазон для демонстрации разницы в производительности
n = 100000
# Традиционный способ
start_time = time.time()
traditional_result = []
for i in range(n):
    if i % 2 == 0:
        traditional_result.append(i ** 2)
traditional_time = time.time() - start_time
# Генератор списков
start_time = time.time()
comprehension_result = [i ** 2 for i in range(n) if i % 2 == 0]
comprehension_time = time.time() - start_time
print(f"Традиционный способ: {traditional_time:.4f} секунд")
print(f"Генератор списков: {comprehension_time:.4f} секунд")
print(f"Ускорение: {traditional_time / comprehension_time:.2f}x")
print(f"Результаты идентичны: {traditional_result == comprehension_result}")
# Сравнение размеров кода
print("\nСравнение читаемости:")
print("Традиционный код:")
print("""    result = []
    for i in range(n):
        if i % 2 == 0:
            result.append(i ** 2)""")
print("\nГенератор списков:")
print("    result = [i ** 2 for i in range(n) if i % 2 == 0]")
```

Ограничения и рекомендации

Несмотря на мощность генераторов списков, важно использовать их разумно:

```
# Хороший пример - простая логика
good_example = [x * 2 for x in range(10) if x % 3 == 0]
# Плохой пример - слишком сложная логика
# Лучше использовать обычный цикл для читаемости
complex_numbers = [1, 2, 3, 4, 5]
bad_example = [
    (x ** 2 + x + 1) if x % 2 == 0 else (x ** 3 - 2 * x) if x % 3 == 0 else x
    for x in complex_numbers
    if x > 1 and (x % 2 == 0 or x % 3 == 0)
]
# Лучше переписать как обычный цикл
better_approach = []
for x in complex_numbers:
    if x > 1 and (x % 2 == 0 or x % 3 == 0):
        if x % 2 == 0:
            result = x ** 2 + x + 1
        elif x % 3 == 0:
            result = x ** 3 - 2 * x
        else:
            result = x
        better_approach.append(result)
print("Сложный генератор списков:", bad_example)
print("Понятный цикл:", better_approach)
print("Результаты идентичны:", bad_example == better_approach)
```

Генераторы списков следует использовать, когда логика остается простой и понятной. Если выражение становится слишком сложным, предпочтение следует отдать традиционным циклам для сохранения читаемости кода.

4.7. Практические алгоритмы: поиск, сортировка

Циклы составляют основу большинства алгоритмов обработки данных. Даже самые базовые алгоритмы, такие как поиск элемента в последовательности или сортировка списка, невозможны без эффективного использования циклов. На этом этапе важно научиться применять полученные знания для решения реальных задач.

Поиск элемента в списке

Обычный линейный поиск (linear search) позволяет найти элемент в последовательности, перебирая ее последовательно с начала до конца.

```
# Задача: Найти элемент target в списке numbers
numbers = [12, 5, 7, 20, 3, 14, 7, 9]
target = 7
found = False
positions = []
for index in range(len(numbers)):
```

```
    if numbers[index] == target:
        found = True
        positions.append(index)
        print(f"Элемент {target} найден в позиции {index}")
if not found:
    print(f"Элемент {target} не найден")
else:
    print(f"Элемент {target} встречается в позициях: {positions}")
```

Этот код демонстрирует, как с помощью цикла можно не только находить первое, но и все вхождения интересующего значения в списке. Такой подход универсален и понятен, однако при больших объемах данных может быть неэффективен — поиск осуществляется за время, пропорциональное длине списка (линейная сложность, `O(n)`).

Поиск максимального/минимального значения

Чтобы найти максимум в последовательности, необязательно использовать встроенную функцию `max()`, важно уметь реализовать алгоритм самостоятельно:

```
# Поиск максимального значения в списке
numbers = [7, 12, 4, 19, 8, 5]
max_value = numbers[0]
for n in numbers:
    if n > max_value:
        max_value = n
print(f"Максимальное значение: {max_value}")
```

Аналогично реализуется поиск минимального значения.

Алгоритмы сортировки: пузырьковая сортировка

Один из самых простых методов сортировки — сортировка пузырьком (bubble sort). Несмотря на невысокую эффективность на больших данных, этот алгоритм отлично подходит для иллюстрации работы вложенных циклов:

```
# Сортировка списка по возрастанию (пузырьковая сортировка)
numbers = [6, 4, 2, 7, 1]
n = len(numbers)
for i in range(n - 1):
    for j in range(n - 1 - i):
        if numbers[j] > numbers[j + 1]:
            # Меняем элементы местами
            numbers[j], numbers[j + 1] = numbers[j + 1], numbers[j]
    print(f"Промежуточный результат после {i + 1} прохода:", numbers)
print("Итоговый отсортированный список:", numbers)
```

На каждом проходе внутреннего цикла «всплывает» очередной наибольший элемент в конец списка, а внешний цикл постепенно сокращает область сравнения.

Поиск подстроки в строке

Часто требуется найти вхождение последовательности символов (подстроки) в строке. Стандартный способ — метод `find()`, но важно понять и принцип перебора:

```
# Поиск всех вхождений подстроки substring в строке text
text = "abacabacaba"
substring = "aba"
positions = []
pos = text.find(substring)
while pos != -1:
    positions.append(pos)
    pos = text.find(substring, pos + 1)
print(f"Подстрока '{substring}' найдена в позициях: {positions}")
```

Этот метод удобно использовать для анализа текста и решения прикладных задач, таких как поиск ключевых слов, обработка данных или построение частотных словарей.

4.8. Большая практическая работа: анализ текста

Чтобы закрепить навыки, свяжем основные концепции главы в одной комплексной задаче. Предположим, стоит задача провести анализ текста: посчитать количество слов, частоту встречаемости каждого слова, выявить наиболее частое, самое длинное и самое короткое слова, а также провести базовую фильтрацию.

Постановка задачи

Написать программу, принимающую на вход текст и выполняющую следующие действия:

- разбивает текст на слова;
- считает общее количество слов;
- ведет подсчет частоты каждого слова;
- определяет самое часто встречающееся слово;
- находит самое короткое и самое длинное слова;
- отбрасывает слова длиной менее трех символов и повторяет анализ.

Пример реализации

```
text = """
Python – популярный язык программирования. Python прост в освоении,
но в то же время мощный и гибкий. Многие программисты выбирают Python
для анализа данных,
разработки веб-приложений, машинного обучения и автоматизации.
"""
# 1. Приведение к нижнему регистру и разделение на слова
words = text.lower().split()
# 2. Очистка от пунктуации
clean_words = []
for word in words:
```

```
    # Удаляем символы пунктуации с концов слова
    word = word.strip("--.,;:!?\"'()[]{}")
    if word:  # Пропускаем пустые строки
        clean_words.append(word)
print("Всего слов:", len(clean_words))
# 3. Подсчет частот слов
word_count = {}
for word in clean_words:
    if word in word_count:
        word_count[word] += 1
    else:
        word_count[word] = 1
# 4. Поиск самого частого слова
max_freq = 0
most_common_word = ""
for word, count in word_count.items():
    if count > max_freq:
        max_freq = count
        most_common_word = word
print(f"Самое частое слово: '{most_common_word}' (встречается {max_freq}
раз)")
# 5. Поиск самого короткого и самого длинного слова
min_length_word = min(clean_words, key=len)
max_length_word = max(clean_words, key=len)
print(f"Самое короткое слово: '{min_length_word}'")
print(f"Самое длинное слово: '{max_length_word}'")
# 6. Повторяем анализ только для слов длиной более двух символов
filtered_words = [word for word in clean_words if len(word) > 2]
print("\nФильтрация: только слова длиной более двух символов")
print("Всего слов:", len(filtered_words))
filtered_word_count = {}
for word in filtered_words:
    if word in filtered_word_count:
        filtered_word_count[word] += 1
    else:
        filtered_word_count[word] = 1
# Самое частое из длинных слов
max_freq = 0
most_common_word = ""
for word, count in filtered_word_count.items():
    if count > max_freq:
        max_freq = count
        most_common_word = word
print(f"Самое частое длинное слово: '{most_common_word}' (встречается {max_
freq} раз)")
if filtered_words:
    min_length_word = min(filtered_words, key=len)
    max_length_word = max(filtered_words, key=len)
    print(f"Самое короткое длинное слово: '{min_length_word}'")
    print(f"Самое длинное слово: '{max_length_word}'")
```

Объяснение и расширение задачи

В этом проекте использованы циклы для обработки последовательностей, подсчета статистики, анализа длины строк, фильтрации и поиска. Код сопровождается выводом промежуточных результатов, чтобы показать протекание анализа.

В качестве расширения задания можно, например, игнорировать «стоп-слова» (то есть часто встречающиеся, но малоинформативные слова), анализировать регистр, строить график частот или сохранять результаты в файл. Основная цель — научиться разрабатывать целостные алгоритмы с помощью базовых средств Python, делая код надежным и читаемым.

Глава 5. Функции — строительные блоки программ

До сих пор мы изучали основные элементы программирования: переменные, условия, циклы. Эти инструменты позволяют создавать программы, но по мере роста их сложности возникает потребность в более мощных средствах организации кода. Функции (functions) становятся тем фундаментальным инструментом, который превращает хаотичный набор инструкций в структурированную, читаемую и поддерживаемую программу.

Функции — это не просто техническая возможность языка программирования. Они воплощают в себе один из важнейших принципов инженерного мышления: разделение сложной задачи на простые, понятные части. Каждая функция решает одну конкретную задачу и может быть использована многократно. Это превращает программирование из ремесла в настоящую инженерную дисциплину.

5.1. Зачем нужны функции: принцип DRY

Представьте, что вы строите дом и вам нужно сделать несколько одинаковых окон. Неразумно было бы каждый раз заново измерять, резать и подгонять каждую деталь. Вместо этого вы создаете шаблон и используете его многократно. В программировании функции играют роль таких шаблонов — они позволяют определить последовательность действий один раз и затем использовать ее столько раз, сколько необходимо.

Проблема дублирования кода

Рассмотрим конкретный пример. Предположим, мы создаем программу для анализа продаж в магазине, и нам нужно рассчитывать налог с продаж для разных товаров:

```
# Расчет налога для первого товара
price1 = 100
tax_rate = 0.12
tax1 = price1 * tax_rate
total1 = price1 + tax1
print(f"Товар 1: цена {price1}, налог {tax1:.2f}, итого {total1:.2f}")
# Расчет налога для второго товара
price2 = 250
tax_rate = 0.12
tax2 = price2 * tax_rate
total2 = price2 + tax2
print(f"Товар 2: цена {price2}, налог {tax2:.2f}, итого {total2:.2f}")
# Расчет налога для третьего товара
price3 = 75
tax_rate = 0.12
tax3 = price3 * tax_rate
total3 = price3 + tax3
print(f"Товар 3: цена {price3}, налог {tax3:.2f}, итого {total3:.2f}")
```

Даже в этом простом примере видно, что логика расчета повторяется трижды. Такой подход создает несколько серьезных проблем.

- **Избыточность кода.**
  Одна и та же логика записана несколько раз, что делает программу длиннее, чем необходимо. В реальных проектах такая избыточность может достигать катастрофических масштабов.
- **Сложность сопровождения.**
  Если налоговая ставка изменится, нам придется найти и исправить все места, где она используется. В большой программе легко пропустить какое-то место, что приведет к ошибке.
- **Высокая вероятность ошибок.**
  При копировании кода легко допустить опечатку или неточность. Чем больше копий, тем выше вероятность ошибки.
- **Снижение читаемости.**
  Повторяющийся код затрудняет понимание основной логики программы, скрывая ее за массой технических деталей.

Принцип DRY: Don't Repeat Yourself

Принцип DRY (Don't Repeat Yourself — «не повторяйся») является одним из основополагающих принципов качественного программирования. Он гласит: каждая часть знаний должна иметь единственное, недвусмысленное, авторитетное представление в системе.

Этот принцип не означает, что в коде не должно быть никаких повторений. Речь идет о повторении логики, алгоритмов, знаний о предметной области. Повторение переменных или простых операций может быть вполне оправданным.

Нарушение принципа DRY часто называют принципом WET (Write Everything Twice — «пиши всё дважды») или «копипаста-программирование». Такой подход ведет к созданию хрупких, трудно поддерживаемых систем.

Функции как решение проблемы повторения

Функции позволяют нам инкапсулировать (заключить в капсулу) повторяющуюся логику и использовать ее многократно. Давайте перепишем наш пример с расчетом налога, используя функцию:

```
def calculate_total_price(price, tax_rate=0.12):
    """Рассчитывает общую стоимость товара с учетом налога"""
    tax = price * tax_rate
    total = price + tax
    return total, tax
# Использование функции для разных товаров
price1 = 100
total1, tax1 = calculate_total_price(price1)
print(f"Товар 1: цена {price1}, налог {tax1:.2f}, итого {total1:.2f}")
price2 = 250
total2, tax2 = calculate_total_price(price2)
print(f"Товар 2: цена {price2}, налог {tax2:.2f}, итого {total2:.2f}")
```

```
price3 = 75
total3, tax3 = calculate_total_price(price3)
print(f"Товар 3: цена {price3}, налог {tax3:.2f}, итого {total3:.2f}")
```

Теперь логика расчета написана только один раз, а используется трижды. Если налоговая ставка изменится, нам нужно будет исправить только одну строку кода.

Преимущества использования функций

- **Повторное использование кода (reusability).**
  Написанную один раз функцию можно использовать в любом месте программы любое количество раз. Это существенно сокращает объем кода и время разработки.
- **Модульность (modularity).**
  Функции позволяют разбить сложную программу на отдельные, логически связанные блоки. Каждый блок решает свою конкретную задачу, что делает программу более понятной.
- **Упрощение отладки (debugging).**
  Когда в программе возникает ошибка, гораздо проще найти и исправить ее в одной функции, чем искать во множестве копий кода.
- **Тестируемость.**
  Функции можно тестировать отдельно от остальной программы, что значительно упрощает процесс обеспечения качества.
- **Абстракция.**
  Функции позволяют скрыть сложные детали реализации за простым интерфейсом. Пользователю функции не нужно знать, как именно она работает внутри.

Функции в реальном мире программирования

В профессиональной разработке принцип DRY применяется повсеместно. Современные программы состоят из тысяч функций, каждая из которых решает свою специфическую задачу. Библиотеки и фреймворки — это, по сути, коллекции функций, созданные для решения типовых задач.

Например, когда вы используете функцию `print()`, вам не нужно думать о том, как именно Python выводит текст на экран. Все сложности работы с операционной системой скрыты внутри этой функции. Вы просто используете ее интерфейс — передаете текст, который нужно вывести.

5.2. Создание собственных функций

Теперь, когда мы понимаем важность функций, давайте научимся создавать их самостоятельно. В Python функции определяются с помощью ключевого слова `def` (от англ. define — «определить»).

Синтаксис определения функции

Базовый синтаксис определения функции в Python выглядит следующим образом:

```
def имя_функции(параметры):
```

```
    """Документация функции (опционально)"""
    тело_функции
    return результат  # опционально
```

Разберем каждый элемент этой конструкции.

- **Ключевое слово** `def` сообщает Python, что далее следует определение функции. Это зарезервированное слово языка, которое нельзя использовать для других целей.
- **Имя функции** должно следовать тем же правилам, что и имена переменных: начинаться с буквы или подчеркивания, содержать только буквы, цифры и подчеркивания. По соглашению Python, имена функций пишутся в нижнем регистре с подчеркиваниями между словами.
- **Параметры** (также называемые аргументами) — это входные данные функции. Они перечисляются в круглых скобках через запятую. Функция может не иметь параметров вовсе.
- **Двоеточие `:`** обязательно завершает строку определения функции, как и в случае с условными конструкциями и циклами.
- **Тело функции** содержит код, который будет выполняться при вызове функции. Обязательно должно быть написано с отступом.
- **Оператор `return`** завершает выполнение функции и возвращает результат. Он не является обязательным.

Простейшие примеры функций

Начнем с самой простой функции — той, которая не принимает параметров и ничего не возвращает:

```
def greet():
    """Выводит приветственное сообщение"""
    print("Добро пожаловать в мир функций!")
    print("Здесь начинается настоящее программирование.")
# Вызов функции
greet()
```

Эта функция демонстрирует базовые принципы: после определения функцию нужно вызвать, чтобы она выполнилась. Вызов происходит путем указания имени функции с круглыми скобками.

Усложним пример, добавив параметр:

```
def greet_user(name):
    """Выводит персональное приветствие пользователю"""
    print(f"Привет, {name}!")
    print("Добро пожаловать в программу.")
# Вызовы функции с разными аргументами
greet_user("Анна")
greet_user("Михаил")
greet_user("Елена")
```

Теперь наша функция стала более универсальной — она может приветствовать любого пользователя, имя которого мы передадим в качестве аргумента.

Функции с возвращаемым значением

Многие функции не только выполняют какие-то действия, но и возвращают результат своей работы. Для этого используется оператор `return`:

```
def square(number):
    """Возвращает квадрат числа"""
    result = number * number
    return result
# Использование функции
x = 5
x_squared = square(x)
print(f"Квадрат числа {x} равен {x_squared}")
# Можно использовать результат сразу в вычислениях
area = square(10)  # area = 100
perimeter = 4 * square(3)  # perimeter = 36
```

Оператор `return` не только возвращает значение, но и немедленно завершает выполнение функции. Код, написанный после `return`, выполняться не будет:

```
def demonstrate_return():
    """Демонстрирует работу оператора return"""
    print("Эта строка выведется")
    return "Результат функции"
    print("А эта строка уже не выведется")  # недостижимый код
result = demonstrate_return()
print(result)
```

Функции с несколькими параметрами

Функции могут принимать любое количество параметров:

```
def calculate_rectangle_area(length, width):
    """Рассчитывает площадь прямоугольника"""
    area = length * width
    return area
def introduce_person(first_name, last_name, age):
    """Выводит информацию о человеке"""
    print(f"Имя: {first_name} {last_name}")
    print(f"Возраст: {age} лет")
# Использование функций
room_area = calculate_rectangle_area(5, 3)
print(f"Площадь комнаты: {room_area} кв.м")
introduce_person("Иван", "Петров", 25)
```

Практический пример: функция для проверки четности

Давайте создадим функцию, которая определяет, является ли число четным. Это хороший пример функции, которая инкапсулирует полезную логику:

```
def is_even(number):
    """
    Проверяет, является ли число четным.
```

```
      Аргументы:
        number: целое число для проверки

    Возвращает:
        True, если число четное, False в противном случае
    """
    return number % 2 == 0
# Примеры использования
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    if is_even(num):
        print(f"Число {num} четное")
    else:
        print(f"Число {num} нечетное")
```

Эта функция демонстрирует важный принцип: хорошая функция делает код более читаемым. Сравните `if is_even(num):` с `if num % 2 == 0:`. Первый вариант гораздо более понятен — он читается как обычный английский текст.

Соглашения об именовании функций

В Python существуют устоявшиеся соглашения о том, как называть функции.

- **Используйте описательные имена.** Имя функции должно ясно указывать на то, что она делает. `calculate_total()` лучше, чем `calc()` или `c()`.
- **Используйте глаголы.** Функции выполняют действия, поэтому их имена должны содержать глаголы: `get_user_input()`, `validate_email()`, `send_message()`.
- **Будьте последовательны.** Если в одной части программы вы используете `get_` как префикс для функций получения данных, используйте его везде.

Документирование функций

Хорошие функции должны быть документированы. Python предоставляет специальный механизм для этого — строки документации (docstrings). Они пишутся в тройных кавычках сразу после определения функции:

```
def fahrenheit_to_celsius(fahrenheit):
    """
    Преобразует температуру из шкалы Фаренгейта в шкалу Цельсия.

    Аргументы:
        fahrenheit (float): Температура в градусах Фаренгейта

    Возвращает:
        float: Температура в градусах Цельсия

    Примеры:
        >>> fahrenheit_to_celsius(32)
        0.0
        >>> fahrenheit_to_celsius(100)
        37.77777777777778
```

```
    """
    celsius = (fahrenheit - 32) * 5 / 9
    return celsius
# Функция с документацией становится самодокументированной
temp_c = fahrenheit_to_celsius(98.6)
print(f"Нормальная температура тела: {temp_c:.1f}°C")
```

Документация делает функцию значительно более полезной для других разработчиков (и для вас самих через некоторое время). Многие среды разработки автоматически показывают документацию функции при ее использовании.

5.3. Параметры и аргументы: передача данных

В предыдущем разделе мы создали несколько простых функций и использовали термины «параметры» и «аргументы» почти взаимозаменяемо. Настало время разобраться в этих понятиях более точно и изучить различные способы передачи данных в функции.

Параметры vs Аргументы: важное различие

Хотя эти термины часто используются как синонимы, между ними есть важное различие.

Параметры (parameters) — это переменные, которые указываются в определении функции. Они являются частью «подписи» функции и определяют, какие данные функция ожидает получить.

Аргументы (arguments) — это конкретные значения, которые передаются функции при ее вызове.

```
def greet_person(name, age):  # name и age - это параметры
    """Приветствует человека, указывая его имя и возраст"""
    print(f"Привет, {name}! Тебе {age} лет.")
# При вызове функции передаем аргументы
greet_person("Мария", 28)  # "Мария" и 28 - это аргументы
greet_person("Алексей", 35)  # "Алексей" и 35 - это аргументы
```

Понимание этого различия помогает более точно обсуждать функции и их использование.

Позиционные аргументы

По умолчанию аргументы передаются в функцию по позиции — первый аргумент соответствует первому параметру, второй аргумент — второму параметру и так далее:

```
def describe_book(title, author, year, pages):
    """Выводит информацию о книге"""
    print(f"Книга: '{title}'")
    print(f"Автор: {author}")
    print(f"Год издания: {year}")
    print(f"Количество страниц: {pages}")
# Порядок аргументов должен соответствовать порядку параметров
describe_book("1984", "Джордж Оруэлл", 1949, 328)
```

Важно соблюдать правильный порядок аргументов. Если мы перепутаем их местами, функция получит неправильные данные:

```
# Неправильный порядок аргументов приведет к странному результату
describe_book("Джордж Оруэлл", "1984", 328, 1949)
# Автором книги окажется "1984", а годом издания — 328
```

Именованные аргументы

Python позволяет указывать аргументы по имени параметра, что делает вызов функции более понятным и безопасным:

```
def describe_book(title, author, year, pages):
    """Выводит информацию о книге"""
    print(f"Книга: '{title}'")
    print(f"Автор: {author}")
    print(f"Год издания: {year}")
    print(f"Количество страниц: {pages}")
# Используем именованные аргументы
describe_book(title="1984",
              author="Джордж Оруэлл",
              year=1949,
              pages=328)
# При использовании именованных аргументов порядок может быть любым
describe_book(author="Лев Толстой",
              pages=1300,
              title="Война и мир",
              year=1869)
```

Именованные аргументы особенно полезны, когда функция имеет много параметров или когда значение параметра не очевидно из контекста.

Смешивание позиционных и именованных аргументов

Python позволяет смешивать позиционные и именованные аргументы, но с важным ограничением — все позиционные аргументы должны идти перед именованными:

```
def register_user(username, email, age, city="Не указан", country="Россия"):
    """Регистрирует нового пользователя"""
    print(f"Пользователь: {username}")
    print(f"Email: {email}")
    print(f"Возраст: {age}")
    print(f"Город: {city}")
    print(f"Страна: {country}")
# Правильно: позиционные аргументы идут первыми
register_user("ivan123", "ivan@example.com", 25, city="Москва")
# Правильно: все аргументы именованные
register_user(username="maria_k",
              email="maria@example.com",
              age=30,
              city="Санкт-Петербург")
# Неправильно: именованный аргумент перед позиционным вызовет ошибку
# register_user(username="alex", "alex@example.com", age=22)  # SyntaxError
```

Практический пример: функция расчета стоимости доставки

Давайте создадим функцию, которая демонстрирует различные способы передачи аргументов:

```
def calculate_shipping_cost(weight, distance, express=False, insurance=False):
    """
    Рассчитывает стоимость доставки.

    Аргументы:
        weight: вес посылки в килограммах
        distance: расстояние в километрах
        express: экспресс-доставка (по умолчанию False)
        insurance: страхование груза (по умолчанию False)
    """
    base_cost = weight * 0.5 + distance * 0.1

    if express:
        base_cost *= 2  # экспресс-доставка удваивает стоимость

    if insurance:
        base_cost += 50  # страховка стоит 50 рублей

    return round(base_cost, 2)
# Различные способы вызова функции
cost1 = calculate_shipping_cost(2.5, 100)  # обычная доставка
cost2 = calculate_shipping_cost(2.5, 100, True)  # экспресс-доставка
cost3 = calculate_shipping_cost(2.5, 100, express=True, insurance=True)
cost4 = calculate_shipping_cost(weight=1.0, distance=50, insurance=True)
print(f"Обычная доставка: {cost1} руб.")
print(f"Экспресс-доставка: {cost2} руб.")
print(f"Экспресс + страховка: {cost3} руб.")
print(f"Со страховкой: {cost4} руб.")
```

5.4. Возвращение значений: результат работы функции

Мы уже видели функции, которые возвращают значения с помощью оператора `return`. Настало время изучить этот механизм подробнее и понять, как функции могут передавать результаты своей работы обратно в вызывающий код.

Оператор `return`

Оператор `return` выполняет две важные функции: он завершает выполнение функции и возвращает значение вызывающему коду. Если в функции нет оператора `return`, она автоматически возвращает специальное значение `None`:

```
def function_without_return():
    """Функция без явного return"""
    print("Эта функция что-то делает...")
    # неявно возвращает None
def function_with_return():
```

```
    """Функция с явным return"""
    print("Эта функция тоже что-то делает...")
    return "Результат работы"
result1 = function_without_return()
result2 = function_with_return()
print(f"Результат первой функции: {result1}")  # None
print(f"Результат второй функции: {result2}")  # Результат работы
```

Возврат различных типов данных

Функции могут возвращать значения любого типа: числа, строки, списки, словари и даже другие функции:

```
def get_user_age():
    """Возвращает возраст пользователя как число"""
    return 25
def get_user_name():
    """Возвращает имя пользователя как строку"""
    return "Анна"
def get_user_hobbies():
    """Возвращает хобби пользователя как список"""
    return ["чтение", "программирование", "путешествия"]
def get_user_info():
    """Возвращает информацию о пользователе как словарь"""
    return {
        "name": "Анна",
        "age": 25,
        "city": "Москва",
        "hobbies": ["чтение", "программирование", "путешествия"]
    }
# Использование функций
age = get_user_age()
name = get_user_name()
hobbies = get_user_hobbies()
info = get_user_info()
print(f"Пользователь {name}, возраст {age}")
print(f"Хобби: {', '.join(hobbies)}")
print(f"Полная информация: {info}")
```

Возврат нескольких значений

Python позволяет возвращать несколько значений одновременно. На самом деле функция возвращает кортеж (`tuple`), но Python автоматически «распаковывает» его:

```
def calculate_circle_properties(radius):
    """Рассчитывает площадь и периметр окружности"""
    import math
    area = math.pi * radius ** 2
    circumference = 2 * math.pi * radius
    return area, circumference  # возвращаем два значения
# Получаем оба значения
```

```
circle_area, circle_circumference = calculate_circle_properties(5)
print(f"Площадь: {circle_area:.2f}")
print(f"Периметр: {circle_circumference:.2f}")
# Можно получить результат как кортеж
result = calculate_circle_properties(3)
print(f"Результат как кортеж: {result}")
print(f"Площадь: {result[0]:.2f}, периметр: {result[1]:.2f}")
```

Условный возврат значений

Функция может возвращать разные значения в зависимости от условий. Важно убедиться, что все возможные пути выполнения функции приводят к возврату значения:

```
def classify_temperature(celsius):
    """Классифицирует температуру по категориям"""
    if celsius < 0:
        return "холодно", "зима"
    elif celsius < 15:
        return "прохладно", "весна/осень"
    elif celsius < 25:
        return "тепло", "лето"
    else:
        return "жарко", "пик лета"
def get_grade_description(score):
    """Возвращает описание оценки или None для недопустимых значений"""
    if score < 0 or score > 100:
        return None  # недопустимое значение
    elif score >= 90:
        return "Отлично"
    elif score >= 80:
        return "Хорошо"
    elif score >= 70:
        return "Удовлетворительно"
    else:
        return "Неудовлетворительно"
# Примеры использования
temp_desc, season = classify_temperature(22)
print(f"22°C: {temp_desc}, сезон: {season}")
grade = get_grade_description(85)
if grade:
    print(f"Оценка 85 баллов: {grade}")
else:
    print("Недопустимое значение баллов")
```

Досрочный возврат

Оператор `return` можно использовать для досрочного завершения функции, что особенно полезно для проверки условий:

```
def divide_numbers(a, b):
    """Делит два числа с проверкой на деление на ноль"""
```

```
    if b == 0:
        return None  # досрочный возврат при ошибке

    return a / b
def validate_email(email):
    """Проверяет корректность email-адреса (упрощенная версия)"""
    if "@" not in email:
        return False  # досрочный возврат при ошибке

    if "." not in email:
        return False  # еще одна проверка

    parts = email.split("@")
    if len(parts) != 2:
        return False

    return True  # все проверки пройдены
# Примеры использования
result1 = divide_numbers(10, 2)
result2 = divide_numbers(10, 0)
print(f"10 / 2 = {result1}")
print(f"10 / 0 = {result2}")
emails = ["user@example.com", "invalid-email", "user@", "@example.com"]
for email in emails:
    is_valid = validate_email(email)
    print(f"{email}: {'корректный' if is_valid else 'некорректный'}")
```

Практический пример: функция анализа текста

Создадим функцию, которая анализирует текст и возвращает несколько характеристик:

```
def analyze_text(text):
    """
    Анализирует текст и возвращает различные статистики.

    Возвращает кортеж: (количество символов, количество слов,
                        количество предложений, самое длинное слово)
    """
    if not text:  # проверка на пустой текст
        return 0, 0, 0, ""

    # Подсчет символов
    char_count = len(text)

    # Подсчет слов
    words = text.split()
    word_count = len(words)

    # Подсчет предложений (упрощенный способ)
    sentence_endings = text.count('.') + text.count('!') + text.count('?')
```

```
    sentence_count = max(1, sentence_endings)  # минимум одно предложение

    # Поиск самого длинного слова
    longest_word = ""
    for word in words:
        # Убираем знаки препинания
        clean_word = word.strip('.,!?;:')
        if len(clean_word) > len(longest_word):
            longest_word = clean_word

    return char_count, word_count, sentence_count, longest_word
# Тестирование функции
sample_text = """
Python - это мощный и универсальный язык программирования.
Он отлично подходит для начинающих разработчиков!
Функции делают код более организованным и читаемым.
"""
chars, words, sentences, longest = analyze_text(sample_text.strip())
print(f"Анализ текста:")
print(f"Символов: {chars}")
print(f"Слов: {words}")
print(f"Предложений: {sentences}")
print(f"Самое длинное слово: '{longest}' ({len(longest)} символов)")
```

5.5. Область видимости переменных

Область видимости (scope) — это одна из важнейших концепций в программировании, определяющая, где в программе доступна та или иная переменная. Понимание областей видимости критически важно для написания корректных программ и избегания трудно обнаруживаемых ошибок.

Локальные переменные

Переменные, созданные внутри функции, называются локальными переменными. Они существуют только во время выполнения функции и недоступны за ее пределами:

```
def calculate_total():
    """Демонстрация локальных переменных"""
    price = 100  # локальная переменная
    tax = 0.12   # локальная переменная
    total = price * (1 + tax)  # локальная переменная
    return total
result = calculate_total()
print(f"Итоговая сумма: {result}")
# Попытка обратиться к локальной переменной вызовет ошибку
# print(price)  # NameError: name 'price' is not defined
```

Локальные переменные имеют несколько важных характеристик:

- **Изоляция.** Локальные переменные изолированы от остальной программы. Изменения внутри функции не влияют на переменные за ее пределами.

- **Время жизни.** Локальные переменные создаются при входе в функцию и уничтожаются при выходе из нее.
- **Производительность.** Доступ к локальным переменным работает быстрее, чем к глобальным.

Глобальные переменные

Переменные, определенные на уровне модуля (вне всех функций), называются глобальными. Они доступны везде в программе:

```
# Глобальная переменная
company_name = "ТехноСофт"
tax_rate = 0.12
def print_company_info():
    """Функция использует глобальную переменную"""
    print(f"Компания: {company_name}")
    print(f"Налоговая ставка: {tax_rate * 100}%")
def calculate_price_with_tax(price):
    """Функция использует глобальную переменную в вычислениях"""
    return price * (1 + tax_rate)
print_company_info()
final_price = calculate_price_with_tax(1000)
print(f"Цена с налогом: {final_price}")
```

Конфликт имен: локальные переменные скрывают глобальные

Если внутри функции создать переменную с тем же именем, что и глобальная переменная, локальная переменная «скроет» глобальную:

```
message = "Глобальное сообщение"
def test_scope():
    message = "Локальное сообщение"  # создаем локальную переменную
    print(f"Внутри функции: {message}")
print(f"До вызова функции: {message}")
test_scope()
print(f"После вызова функции: {message}")
```

Результат выполнения:

```
До вызова функции: Глобальное сообщение
Внутри функции: Локальное сообщение
После вызова функции: Глобальное сообщение
```

Ключевое слово `global`

Если нужно изменить глобальную переменную внутри функции, используется ключевое слово `global`:

```
counter = 0  # глобальная переменная
def increment_counter():
    global counter  # указываем, что хотим работать с глобальной переменной
    counter += 1
```

```
    print(f"Счетчик увеличен до {counter}")
def reset_counter():
    global counter
    counter = 0
    print("Счетчик сброшен")
print(f"Начальное значение: {counter}")
increment_counter()
increment_counter()
increment_counter()
reset_counter()
```

Важное предупреждение: использование глобальных переменных следует минимизировать, так как они делают программу менее предсказуемой и сложной для понимания.

Параметры функции — это локальные переменные

Параметры функции автоматически становятся локальными переменными:

```
name = "Глобальное имя"
def greet(name):  # параметр name - это локальная переменная
    print(f"Привет, {name}!")
    name = name.upper()  # изменяем локальную копию
    print(f"Измененное имя: {name}")
print(f"До вызова функции: {name}")
greet("Анна")
print(f"После вызова функции: {name}")
```

Практический пример: управление банковским счетом

Давайте создадим пример, который демонстрирует правильное использование областей видимости:

```
# Глобальные переменные для хранения состояния счета
account_balance = 1000.0
account_holder = "Иван Петров"
transaction_count = 0
def get_balance():
    """Возвращает текущий баланс счета"""
    return account_balance
def deposit(amount):
    """Пополняет счет на указанную сумму"""
    global account_balance, transaction_count

    if amount <= 0:
        print("Сумма пополнения должна быть положительной")
        return False

    account_balance += amount
    transaction_count += 1
   print(f"Счет пополнен на {amount:.2f} руб. Баланс: {account_balance:.2f}
руб.")
```

```
    return True
def withdraw(amount):
    """Снимает указанную сумму со счета"""
    global account_balance, transaction_count

    if amount <= 0:
        print("Сумма снятия должна быть положительной")
        return False

    if amount > account_balance:
        print(f"Недостаточно средств. Баланс: {account_balance:.2f} руб.")
        return False

    account_balance -= amount
    transaction_count += 1
    print(f"Со счёта снято {amount:.2f} руб. Баланс: {account_balance:.2f}
руб.")
    return True
def get_account_info():
    """Выводит информацию о счете"""
    print(f"Владелец счета: {account_holder}")
    print(f"Текущий баланс: {account_balance:.2f} руб.")
    print(f"Количество операций: {transaction_count}")
# Использование функций
get_account_info()
print("\n--- Операции со счетом ---")
deposit(500)
withdraw(200)
withdraw(2000)  # попытка снять больше, чем есть на счете
deposit(-100)   # попытка пополнить на отрицательную сумму
print("\n--- Итоговая информация ---")
get_account_info()
```

Этот пример показывает, как глобальные переменные могут быть полезны для хранения состояния программы, но их использование должно быть тщательно продумано и ограничено.

Рекомендации по работе с областями видимости

- **Предпочитайте локальные переменные.** Они делают функции более предсказуемыми и легкими для тестирования.
- **Минимизируйте использование глобальных переменных.** Передавайте необходимые данные через параметры функций.
- **Используйте выразительные имена.** Хорошие имена переменных снижают вероятность конфликтов имен.
- **Группируйте связанные данные.** Вместо множества глобальных переменных рассмотрите возможность использования структур данных.

Понимание областей видимости — это основа для написания чистого, понятного и надежного кода. В следующих разделах мы изучим еще более мощные возможности функций, которые помогут нам создавать еще более элегантные программы.

5.6. Значения по умолчанию и именованные параметры

В реальном программировании часто возникают ситуации, когда некоторые параметры функции имеют естественные значения, которые подходят в большинстве случаев. Python предоставляет элегантный механизм для работы с такими ситуациями — значения параметров по умолчанию.

Параметры со значениями по умолчанию

Значения по умолчанию позволяют сделать некоторые параметры функции необязательными. Если при вызове функции для такого параметра не указан аргумент, будет использовано значение по умолчанию:

```
def greet_user(name, greeting="Привет", punctuation="!"):
    """
    Приветствует пользователя с настраиваемым сообщением.

    Аргументы:
        name: имя пользователя (обязательный)
        greeting: приветствие (по умолчанию "Привет")
        punctuation: знак препинания (по умолчанию "!")
    """
    message = f"{greeting}, {name}{punctuation}"
    print(message)
# Различные способы вызова функции
greet_user("Анна")  # используются все значения по умолчанию
greet_user("Михаил", "Добро пожаловать")  # изменяем приветствие
greet_user("Елена", "Здравствуйте", ".")  # изменяем всё
```

Результат выполнения:

```
Привет, Анна!
Добро пожаловать, Михаил!
Здравствуйте, Елена.
```

Правила использования значений по умолчанию

Порядок параметров. Параметры со значениями по умолчанию должны следовать после обязательных параметров:

```
# Правильно
def create_user_profile(name, age, city="Не указан", country="Россия"):
    pass
# Неправильно - вызовет SyntaxError
# def create_user_profile(name="Аноним", age, city="Не указан"):
#     pass
```

Значения по умолчанию вычисляются один раз. Это важная особенность Python, которая может привести к неожиданному поведению:

```
# Проблемный код - НЕ ДЕЛАЙТЕ ТАК!
```

```
def add_item_to_list(item, target_list=[]):
    """ОПАСНАЯ функция - список создается только один раз!"""
    target_list.append(item)
    return target_list
# При каждом вызове используется один и тот же список!
list1 = add_item_to_list("первый")
list2 = add_item_to_list("второй")
print(list1)  # ['первый', 'второй'] - неожиданно!
print(list2)  # ['первый', 'второй'] - тот же список!
```

Правильное решение:

```
def add_item_to_list(item, target_list=None):
    """Безопасная версия функции"""
    if target_list is None:
        target_list = []  # создаем новый список при каждом вызове
    target_list.append(item)
    return target_list
# Теперь всё работает как ожидается
list1 = add_item_to_list("первый")
list2 = add_item_to_list("второй")
print(list1)  # ['первый']
print(list2)  # ['второй']
```

Именованные параметры в сочетании со значениями по умолчанию

Именованные параметры особенно полезны при работе с функциями, имеющими много параметров со значениями по умолчанию:

```
def format_price(amount, currency="руб",
                 include_currency=True, decimal_places=2,
                 thousands_separator=" "):
    """
    Форматирует цену с различными опциями.

    Аргументы:
        amount: сумма для форматирования
        currency: валюта
        include_currency: включать ли обозначение валюты
        decimal_places: количество знаков после точки
        thousands_separator: разделитель тысяч
    """
    # Форматируем число с нужным количеством знаков после точки
    formatted_amount = f"{amount:.{decimal_places}f}"

    # Добавляем разделитель тысяч (упрощенная реализация)
    parts = formatted_amount.split('.')
    integer_part = parts[0]
    decimal_part = parts[1] if len(parts) > 1 else ""

    # Добавляем разделители тысяч
    if len(integer_part) > 3:
```

```
        # Простая реализация для демонстрации
        reversed_digits = integer_part[::-1]
         groups = [reversed_digits[i:i+3] for i in range(0, len(reversed_
digits), 3)]
        integer_part = thousands_separator.join(groups)[::-1]

    result = integer_part
    if decimal_part:
        result += "." + decimal_part

    if include_currency:
        result += " " + currency

    return result
# Различные способы использования
print(format_price(1234.56))  # все значения по умолчанию
print(format_price(1234.56, currency="$"))  # только валюта изменена
print(format_price(1234.56, decimal_places=0))  # без копеек
print(format_price(1234.56, include_currency=False, decimal_places=1))
print(format_price(amount=1234567.89,             thousands_separator=",",
currency="€"))
```

Практический пример: функция подключения к базе данных

Рассмотрим реалистичный пример функции для подключения к базе данных:

```
def connect_to_database(host, database, username, password=None,
                        port=5432, timeout=30, ssl_mode="prefer"):
    """
    Устанавливает подключение к базе данных.

    Аргументы:
        host: адрес сервера базы данных
        database: имя базы данных
        username: имя пользователя
        password: пароль (если None, будет запрошен)
        port: порт подключения (по умолчанию 5432 для PostgreSQL)
        timeout: время ожидания подключения в секундах
        ssl_mode: режим SSL-соединения
    """
    # Если пароль не указан, запрашиваем его
    if password is None:
        import getpass
        password = getpass.getpass(f"Введите пароль для {username}: ")

    # Формируем строку подключения
    connection_string = (
        f"host={host} port={port} dbname={database} "
        f"user={username} password=*** sslmode={ssl_mode} "
        f"connect_timeout={timeout}"
    )
```

```
    print(f"Подключение к базе данных...")
    print(f"Параметры: {connection_string}")
    print("Подключение установлено успешно")

    # В реальной функции здесь был бы код подключения к БД
    return {
        "host": host,
        "database": database,
        "username": username,
        "port": port,
        "connected": True
    }
# Различные способы вызова
connection1 = connect_to_database("localhost", "myapp", "developer")
connection2 = connect_to_database(
    host="production.db.company.com",
    database="main_db",
    username="app_user",
    password="secret_password",
    port=3306,  # MySQL порт
    ssl_mode="require"
)
# Использование только именованных параметров для ясности
connection3 = connect_to_database(
    host="test.db.company.com",
    database="test_db",
    username="test_user",
    timeout=60,
    ssl_mode="disable"
)
```

5.7. Функции как объекты первого класса

В Python функции являются объектами первого класса (first-class objects). Это означает, что функции можно использовать так же, как и любые другие объекты: присваивать переменным, передавать как аргументы другим функциям, возвращать из функций и хранить в структурах данных. Эта особенность открывает мощные возможности для создания гибких и элегантных программ.

Присваивание функций переменным

Функцию можно присвоить переменной, после чего эту переменную можно использовать для вызова функции:

```
def greet_formally(name):
    """Формальное приветствие"""
    return f"Здравствуйте, {name}!"
def greet_casually(name):
    """Неформальное приветствие"""
    return f"Привет, {name}!"
```

```
# Присваиваем функции переменным
formal_greeting = greet_formally
casual_greeting = greet_casually
# Используем переменные как функции
print(formal_greeting("Анна Ивановна"))  # Здравствуйте, Анна Ивановна!
print(casual_greeting("Петя"))  # Привет, Петя!
# Можем даже проверить, что это одна и та же функция
print(formal_greeting is greet_formally)  # True
```

Передача функций как аргументов

Функции можно передавать другим функциям в качестве аргументов. Это мощный механизм для создания гибких и переиспользуемых функций:

```
def apply_to_numbers(numbers, operation):
    """
    Применяет операцию к каждому числу в списке.

    Аргументы:
        numbers: список чисел
        operation: функция, которую нужно применить к каждому числу
    """
    result = []
    for number in numbers:
        result.append(operation(number))
    return result
def square(x):
    """Возводит число в квадрат"""
    return x ** 2
def cube(x):
    """Возводит число в куб"""
    return x ** 3
def double(x):
    """Удваивает число"""
    return x * 2
# Список чисел для обработки
numbers = [1, 2, 3, 4, 5]
# Применяем разные операции
squares = apply_to_numbers(numbers, square)
cubes = apply_to_numbers(numbers, cube)
doubled = apply_to_numbers(numbers, double)
print(f"Исходные числа: {numbers}")
print(f"Квадраты: {squares}")
print(f"Кубы: {cubes}")
print(f"Удвоенные: {doubled}")
```

Функции высшего порядка

Функции высшего порядка (higher-order functions) — это функции, которые принимают другие функции в качестве аргументов или возвращают функции. Они позволяют создавать очень гибкие и мощные абстракции:

```
def create_multiplier(factor):
    """
    Создает функцию, которая умножает число на заданный коэффициент.

    Возвращает функцию, которая умножает свое число на factor.
    """
    def multiplier(x):
        return x * factor

    return multiplier
def create_validator(min_value, max_value):
    """
    Создает функцию для проверки, что число находится в заданном диапазоне.
    """
    def validator(value):
        return min_value <= value <= max_value

    return validator
# Создаем специализированные функции
double = create_multiplier(2)
triple = create_multiplier(3)
multiply_by_ten = create_multiplier(10)
# Используем созданные функции
print(double(5))  # 10
print(triple(4))  # 12
print(multiply_by_ten(7))  # 70
# Создаем валидаторы для разных диапазонов
age_validator = create_validator(0, 150)
percentage_validator = create_validator(0, 100)
print(age_validator(25))  # True
print(age_validator(-5))  # False
print(percentage_validator(85))  # True
print(percentage_validator(150))  # False
```

Хранение функций в структурах данных

Функции можно хранить в списках, словарях и других структурах данных:

```
def add(a, b):
    """Сложение"""
    return a + b
def subtract(a, b):
    """Вычитание"""
    return a - b
def multiply(a, b):
    """Умножение"""
    return a * b
def divide(a, b):
    """Деление"""
    if b != 0:
        return a / b
```

```
    else:
        return "Ошибка: деление на ноль"
# Сохраняем функции в словаре
operations = {
    "+": add,
    "-": subtract,
    "*": multiply,
    "/": divide
}
# Создаем простой калькулятор
def calculator(a, operator, b):
    """Выполняет математическую операцию"""
    if operator in operations:
        operation_function = operations[operator]
        return operation_function(a, b)
    else:
        return "Неизвестная операция"
# Используем калькулятор
print(calculator(10, "+", 5))  # 15
print(calculator(10, "-", 3))  # 7
print(calculator(4, "*", 6))   # 24
print(calculator(15, "/", 3))  # 5.0
print(calculator(10, "%", 3))  # Неизвестная операция
```

Практический пример: система обработки событий

Рассмотрим более сложный пример — простую систему обработки событий:

```
def create_event_system():
    """Создает систему обработки событий"""
    event_handlers = {}  # словарь для хранения обработчиков событий

    def register_handler(event_type, handler_function):
        """Регистрирует обработчик для определенного типа события"""
        if event_type not in event_handlers:
            event_handlers[event_type] = []
        event_handlers[event_type].append(handler_function)
        print(f"Зарегистрирован обработчик для события '{event_type}'")

    def trigger_event(event_type, event_data=None):
        """Вызывает все обработчики для определенного типа события"""
        if event_type in event_handlers:
            print(f"Обрабатывается событие '{event_type}'")
            for handler in event_handlers[event_type]:
                handler(event_data)
        else:
            print(f"Нет обработчиков для события '{event_type}'")

    def list_events():
        """Показывает все зарегистрированные типы событий"""
        print("Зарегистрированные события:")
```

```
        for event_type, handlers in event_handlers.items():
            print(f"  {event_type}: {len(handlers)} обработчик(ов)")

    return register_handler, trigger_event, list_events
# Создаем систему событий
register_handler, trigger_event, list_events = create_event_system()
# Определяем обработчики событий
def on_user_login(user_data):
    """Обработчик входа пользователя в систему"""
    print(f"Пользователь {user_data['username']} вошел в систему")
def on_user_logout(user_data):
    """Обработчик выхода пользователя из системы"""
    print(f"Пользователь {user_data['username']} вышел из системы")
def log_user_activity(user_data):
    """Записывает активность пользователя в лог"""
    print(f"Лог: активность пользователя {user_data['username']}")
def send_welcome_email(user_data):
    """Отправляет приветственное письмо"""
    print(f"Отправлено приветственное письмо на {user_data['email']}")
# Регистрируем обработчики
register_handler("user_login", on_user_login)
register_handler("user_login", log_user_activity)
register_handler("user_login", send_welcome_email)
register_handler("user_logout", on_user_logout)
register_handler("user_logout", log_user_activity)
# Показываем зарегистрированные события
list_events()
# Генерируем события
user_data = {
    "username": "anna_k",
    "email": "anna@example.com",
    "user_id": 12345
}
print("\n--- Событие: вход пользователя ---")
trigger_event("user_login", user_data)
print("\n--- Событие: выход пользователя ---")
trigger_event("user_logout", user_data)
print("\n--- Неизвестное событие ---")
trigger_event("user_registration", user_data)
```

5.8. Рекурсия: функции, вызывающие сами себя

Рекурсия — это один из самых элегантных и мощных инструментов в программировании. Рекурсивная функция решает задачу, вызывая саму себя с упрощенной версией той же задачи. Хотя поначалу рекурсия может показаться сложной концепцией, она естественным образом решает многие задачи и делает код более читаемым.

Что такое рекурсия

Рекурсия основана на принципе «разделяй и властвуй»: сложная задача разбивается на более простые подзадачи того же типа. Каждая рекурсивная функция должна иметь два обязательных элемента.

- **Базовый случай (base case)** — условие, при котором функция прекращает вызывать сама себя и возвращает конкретный результат.
- **Рекурсивный случай (recursive case)** — функция вызывает саму себя с измененными параметрами, приближаясь к базовому случаю.

Простейший пример: факториал

Факториал числа `n` (обозначается `n!`) — это произведение всех натуральных чисел от 1 до `n`. Математическое определение факториала уже содержит рекурсию:

- `0! = 1` (базовый случай)
- `n! = n × (n-1)!` для `n > 0` (рекурсивный случай)

```
def factorial(n):
    """
    Вычисляет факториал числа n рекурсивным способом.

    Аргументы:
        n: неотрицательное целое число

    Возвращает:
        факториал числа n
    """
    # Базовый случай
    if n == 0 or n == 1:
        return 1

    # Рекурсивный случай
    return n * factorial(n - 1)
# Примеры использования
for i in range(6):
    print(f"{i}! = {factorial(i)}")
```

Проследим выполнение `factorial(4)`:

- `factorial(4)` возвращает `4 * factorial(3)`
- `factorial(3)` возвращает `3 * factorial(2)`
- `factorial(2)` возвращает `2 * factorial(1)`
- `factorial(1)` возвращает 1 (базовый случай)
- Результат: `4 * 3 * 2 * 1 = 24`

Рекурсивный обход структур данных

Рекурсия особенно полезна для работы с вложенными структурами данных. Рассмотрим функцию для подсчета суммы всех чисел в произвольно вложенном списке:

```
def sum_nested_list(data):
```

```
    """
    Рекурсивно вычисляет сумму всех чисел в произвольно вложенном списке.

    Аргументы:
        data: число, список или произвольно вложенная структура

    Возвращает:
        сумма всех чисел в структуре данных
    """
    # Базовый случай: если это число
    if isinstance(data, (int, float)):
        return data

    # Базовый случай: если это пустой список
    if not data:
        return 0

    # Рекурсивный случай: если это список
    if isinstance(data, list):
        total = 0
        for item in data:
            total += sum_nested_list(item)  # рекурсивный вызов для каждо-
го элемента
        return total

    # Если это не число и не список, считаем как 0
    return 0
# Примеры использования
test_data = [
    1,
    [2, 3],
    [4, [5, 6]],
    [[7, 8], 9],
    [[[10]]]
]
print(f"Сумма элементов: {sum_nested_list(test_data)}")  # 55
# Более сложный пример
complex_data = [1, [2, [3, [4, [5]]]], 6, [[7, 8], [9, 10]]]
print(f"Сумма сложной структуры: {sum_nested_list(complex_data)}")  # 55
```

Числа Фибоначчи

Последовательность Фибоначчи — классический пример рекурсивных вычислений:

```
def fibonacci_naive(n):
    """
    Наивная рекурсивная реализация чисел Фибоначчи.
    ВНИМАНИЕ: неэффективна для больших n!
    """
    # Базовые случаи
```

```
    if n <= 1:
        return n

    # Рекурсивный случай
    return fibonacci_naive(n - 1) + fibonacci_naive(n - 2)
def fibonacci_optimized(n, memo=None):
    """
    Оптимизированная версия с мемоизацией (запоминанием результатов).
    """
    if memo is None:
        memo = {}  # словарь для хранения уже вычисленных значений

    # Если результат уже вычислен, возвращаем его
    if n in memo:
        return memo[n]

    # Базовые случаи
    if n <= 1:
        memo[n] = n
        return n

    # Рекурсивный случай с сохранением результата
    result = fibonacci_optimized(n - 1, memo) + fibonacci_optimized(n - 2,
memo)
    memo[n] = result
    return result
# Сравнение производительности
import time
n = 35
# Наивная реализация (медленная)
start_time = time.time()
result_naive = fibonacci_naive(n)
naive_time = time.time() - start_time
# Оптимизированная реализация (быстрая)
start_time = time.time()
result_optimized = fibonacci_optimized(n)
optimized_time = time.time() - start_time
print(f"Fibonacci({n}):")
print(f"Наивная реализация: {result_naive} (время: {naive_time:.4f} сек)")
print(f"Оптимизированная: {result_optimized} (время: {optimized_time:.4f}
сек)")
print(f"Ускорение в {naive_time / optimized_time:.0f} раз")
```

Рекурсивный поиск файлов

Рассмотрим практический пример — рекурсивный поиск файлов в директории:

```
def find_files_recursively(directory_structure, target_extension):
    """
    Рекурсивно ищет файлы с определенным расширением в структуре директорий.
```

```
    Аргументы:
        directory_structure: словарь, представляющий структуру директорий
        target_extension: расширение файла для поиска

    Возвращает:
        список найденных файлов
    """
    found_files = []

    for name, content in directory_structure.items():
        if isinstance(content, dict):
            # Это поддиректория, рекурсивно ищем в ней
              subdirectory_files = find_files_recursively(content, target_
extension)
            # Добавляем путь к каждому найденному файлу
            for file in subdirectory_files:
                found_files.append(f"{name}/{file}")
        elif isinstance(content, str) and content == "file":
            # Это файл, проверяем расширение
            if name.endswith(target_extension):
                found_files.append(name)

    return found_files
# Моделируем структуру файловой системы
file_system = {
    "documents": {
        "report.txt": "file",
        "data.csv": "file",
        "presentations": {
            "slides.pptx": "file",
            "notes.txt": "file"
        }
    },
    "images": {
        "photo1.jpg": "file",
        "photo2.png": "file",
        "thumbnails": {
            "thumb1.jpg": "file",
            "thumb2.png": "file"
        }
    },
    "config.txt": "file",
    "readme.md": "file"
}
# Ищем все текстовые файлы
txt_files = find_files_recursively(file_system, ".txt")
print("Найденные .txt файлы:")
for file in txt_files:
    print(f"  {file}")
# Ищем все изображения
jpg_files = find_files_recursively(file_system, ".jpg")
```

```
print("\nНайденные .jpg файлы:")
for file in jpg_files:
    print(f"  {file}")
```

Важные принципы работы с рекурсией

- **Всегда определяйте базовый случай.** Без базового случая рекурсия будет продолжаться бесконечно, что приведет к переполнению стека вызовов.
- **Убедитесь, что рекурсия сходится.** Каждый рекурсивный вызов должен приближать вас к базовому случаю.
- **Учитывайте эффективность.** Наивная рекурсия может быть очень медленной. Рассмотрите использование мемоизации или итерационных решений.
- **Проверьте ограничения глубины рекурсии.** Python имеет ограничение на глубину рекурсии (обычно около 1000 уровней).

Когда использовать рекурсию

- **Структуры данных естественно рекурсивны.** Деревья, вложенные списки, файловые системы — все они идеально подходят для рекурсивной обработки.
- **Задача может быть разложена на подзадачи.** Если большую задачу можно разбить на несколько меньших задач того же типа, рекурсия будет естественным решением.
- **Код становится более читаемым.** Рекурсивное решение часто короче и понятнее итерационного.

Однако помните, что рекурсия не всегда является лучшим решением. Для простых задач итерационные решения могут быть более эффективными и понятными.

5.9. Практический проект: библиотека математических функций

Завершающий проект главы о функциях — это создание собственной библиотеки математических функций. Этот проект позволит вам применить все изученные концепции: создание функций, работу с параметрами, возвращение значений, обработку различных случаев и даже рекурсию.

Задание: создайте библиотеку математических функций `mathlib.py`, которая будет содержать полезные математические вычисления. Ваша библиотека должна включать функции для работы с числами, статистикой и геометрией.

Структура проекта

Разделите свою библиотеку на несколько категорий функций. Начните с базовых математических операций, которые расширяют стандартные возможности Python:

```
# mathlib.py - Библиотека математических функций
def absolute_value(number):
    """
    Вычисляет абсолютное значение числа без использования встроенной функции abs()
    """
```

```
    if number >= 0:
        return number
    else:
        return -number
def power(base, exponent):
    """
    Возводит число в степень без использования оператора ** или функции pow()
    """
    if exponent == 0:
        return 1

    result = 1
    positive_exp = absolute_value(exponent)

    for i in range(positive_exp):
        result = result * base

    if exponent < 0:
        return 1 / result
    else:
        return result
def square_root(number, precision=0.0001):
    """
    Вычисляет квадратный корень методом Ньютона
    """
    if number < 0:
        return None  # Возвращаем None для отрицательных чисел
    if number == 0:
        return 0

    # Начальное приближение
    guess = number / 2

    while True:
        better_guess = (guess + number / guess) / 2
        if absolute_value(guess - better_guess) < precision:
            return better_guess
        guess = better_guess
```

Эти функции демонстрируют важные принципы программирования. Функция `absolute_value` показывает, как создать простую, но полезную утилиту. Функция `power` использует циклы для многократного умножения, а также обрабатывает особые случаи (нулевая и отрицательная степени). Функция `square_root` реализует итеративный алгоритм с заданной точностью.

Рекомендации по реализации

1. Начните с планирования. Определите, какие категории функций вы хотите включить. Рекомендуемые категории:

- Базовые операции (степень, корень, модуль).

- Статистические функции (среднее, медиана, стандартное отклонение).
- Геометрические функции (площади, периметры, объемы).
- Функции для работы с последовательностями (факториал, числа Фибоначчи).

2. Обработка ошибок. Продумайте, как ваши функции должны вести себя при некорректных данных. Используйте возврат специальных значений (`None`, `-1`) или логические проверки.

3. Статистические функции. Создайте набор функций для анализа числовых данных.

4. Создание тестовой программы. Создайте отдельный файл `test_mathlib.py` для проверки ваших функций.

Рекомендации по дальнейшему развитию

Документация. Добавьте подробные комментарии и описания к каждой функции. Укажите типы параметров, возможные ошибки и примеры использования.

Константы. Вынесите часто используемые константы (например π) в отдельные переменные в начале файла.

Валидация входных данных. Создайте вспомогательные функции для проверки корректности данных.

Этот проект демонстрирует практическое применение всех концепций, изученных в главе о функциях. Вы создаете модульный, переиспользуемый код, учитесь обрабатывать граничные случаи и строить программы из небольших, понятных компонентов. Такой подход является фундаментальным для написания качественного программного обеспечения.

Глава 6. Структуры данных Python

До сих пор мы работали с отдельными значениями — числами, строками, булевыми значениями. Но реальные программы редко оперируют изолированными данными. Представьте, что вам нужно обработать список студентов в группе, каталог товаров в магазине или результаты научного эксперимента. Для таких задач Python предоставляет мощные структуры данных — коллекции, которые позволяют организовывать и эффективно обрабатывать группы связанных значений.

Структуры данных (data structures) — это способы организации и хранения данных, которые определяют, как мы можем получать доступ к информации и изменять ее. Выбор правильной структуры данных часто определяет не только читаемость кода, но и производительность программы. Python предлагает четыре основные встроенные структуры данных: списки, кортежи, словари и множества. Каждая из них имеет свои особенности, преимущества и области применения.

6.1. Списки: упорядоченные коллекции данных

Список (list) — это упорядоченная изменяемая коллекция элементов, которая может содержать данные разных типов. Списки являются одной из самых универсальных и часто используемых структур данных в Python. Они позволяют хранить последовательность элементов, сохраняя порядок их добавления и предоставляя быстрый доступ к любому элементу по его позиции.

Создание списков

Список создается с помощью квадратных скобок, элементы разделяются запятыми:

```
# Пустой список
empty_list = []
# Список чисел
numbers = [1, 2, 3, 4, 5]
# Список строк
cities = ["Moscow", "Saint Petersburg", "Novosibirsk", "Yekaterinburg"]
# Смешанный список
mixed_data = [42, "Hello", 3.14, True, "World"]
# Многострочное создание для читаемости
students = [
    "Иван Петров",
    "Мария Сидорова",
    "Алексей Иванов",
    "Елена Козлова"
]
```

Python позволяет создавать списки и другими способами. Функция `list()` может преобразовать любую итерируемую последовательность в список:

```
# Создание списка из строки
letters = list("Python")  # ['P', 'y', 't', 'h', 'o', 'n']
# Создание списка из диапазона
range_list = list(range(1, 6))  # [1, 2, 3, 4, 5]
# Создание списка заданного размера с одинаковыми элементами
zeros = [0] * 5  # [0, 0, 0, 0, 0]
```

Индексация и доступ к элементам

Каждый элемент списка имеет свой индекс (index) — числовую позицию, начинающуюся с нуля. Python поддерживает как положительную индексацию (слева направо), так и отрицательную (справа налево):

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]
# Положительная индексация (с начала)
print(fruits[0])   # apple
print(fruits[1])   # banana
print(fruits[4])   # elderberry
# Отрицательная индексация (с конца)
print(fruits[-1])  # elderberry (последний элемент)
print(fruits[-2])  # date (предпоследний элемент)
print(fruits[-5])  # apple (первый элемент)
# Получение длины списка
print(len(fruits))  # 5
Попытка обратиться к несуществующему индексу приводит к ошибке IndexError:
numbers = [10, 20, 30]
# print(numbers[5])  # IndexError: list index out of range
```

Изменяемость списков

Списки в Python являются изменяемыми (mutable), что означает возможность модификации их содержимого после создания. Это фундаментальное отличие от строк, которые неизменяемы:

```
scores = [85, 92, 78, 94, 88]
# Изменение отдельного элемента
scores[2] = 82  # Заменяем 78 на 82
print(scores)   # [85, 92, 82, 94, 88]
# Изменение нескольких элементов
scores[1:3] = [95, 87]  # Заменяем элементы с индексами 1 и 2
print(scores)   # [85, 95, 87, 94, 88]
```

Особенности работы со ссылками

Важно понимать, что переменная, содержащая список, фактически хранит ссылку на объект в памяти. Это создает интересные эффекты при копировании:

```
original = [1, 2, 3]
reference = original  # Создаем ссылку на тот же объект
reference[0] = 999
print(original)  # [999, 2, 3] - оригинал тоже изменился!
```

```
# Для создания независимой копии используем copy()
independent = original.copy()
independent[1] = 888
print(original)     # [999, 2, 3] - оригинал не изменился
print(independent)  # [999, 888, 3]
```

Списки в условных выражениях

Списки можно использовать в логических контекстах. Пустой список считается `False`, непустой — `True`:

```
empty_cart = []
shopping_cart = ["milk", "bread", "eggs"]
if not empty_cart:
    print("Корзина пуста")  # Это сообщение будет выведено
if shopping_cart:
    print(f"В корзине {len(shopping_cart)} товара")  # И это тоже
```

Вложенные списки

Списки могут содержать другие списки, создавая многомерные структуры данных. Это особенно полезно для представления матриц, таблиц и других сложных структур:

```
# Двумерный список - матрица 3x3
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
# Доступ к элементам вложенного списка
print(matrix[1][2])  # 6 (второй список, третий элемент)
# Представление данных о студентах
students_data = [
    ["Иван Петров", 20, [5, 4, 5, 3]],      # имя, возраст, оценки
    ["Мария Сидорова", 19, [5, 5, 4, 5]],
    ["Алексей Иванов", 21, [4, 3, 4, 4]]
]
# Получение оценок первого студента
first_student_grades = students_data[0][2]
print(first_student_grades)  # [5, 4, 5, 3]
```

6.2. Операции со списками: добавление, удаление, поиск

Списки предоставляют богатый набор методов для манипулирования данными. Понимание этих операций и их эффективности критически важно для написания качественных программ.

Добавление элементов

Python предлагает несколько способов добавления элементов в список, каждый из которых оптимален для определенных сценариев:

- **Метод `append()` — добавление в конец**

Метод `append()` добавляет один элемент в конец списка. Это наиболее эффективная операция добавления с точки зрения производительности:

```
numbers = [1, 2, 3]
numbers.append(4)
print(numbers)  # [1, 2, 3, 4]
# Добавление может быть любого типа
mixed_list = ["hello"]
mixed_list.append(42)
mixed_list.append([10, 20])  # Добавляем весь список как один элемент
print(mixed_list)  # ['hello', 42, [10, 20]]
```

- **Метод `insert()` — добавление по индексу**

Метод `insert()` позволяет вставить элемент в любую позицию списка:

```
colors = ["red", "blue", "green"]
colors.insert(1, "yellow")  # Вставляем "yellow" на позицию 1
print(colors)  # ['red', 'yellow', 'blue', 'green']
# Вставка в начало списка
colors.insert(0, "purple")
print(colors)  # ['purple', 'red', 'yellow', 'blue', 'green']
# Вставка за пределы списка добавляет в конец
colors.insert(100, "orange")
print(colors)  # ['purple', 'red', 'yellow', 'blue', 'green', 'orange']
```

- **Метод `extend()` — добавление нескольких элементов**

Метод `extend()` добавляет все элементы из другой итерируемой последовательности:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
# extend() добавляет элементы, а не список целиком
list1.extend(list2)
print(list1)  # [1, 2, 3, 4, 5, 6]
# Можно расширять строкой
letters = ['a', 'b']
letters.extend("cdef")
print(letters)  # ['a', 'b', 'c', 'd', 'e', 'f']
# Сравните с append()
list3 = [1, 2, 3]
list3.append([4, 5, 6])  # Добавляется весь список как элемент
print(list3)  # [1, 2, 3, [4, 5, 6]]
```

- **Оператор конкатенации `+`**

Оператор `+` создает новый список, объединяя существующие:

```
first_half = [1, 2, 3]
second_half = [4, 5, 6]
complete = first_half + second_half
print(complete)  # [1, 2, 3, 4, 5, 6]
# Исходные списки остаются неизменными
print(first_half)   # [1, 2, 3]
print(second_half)  # [4, 5, 6]
# Оператор += изменяет исходный список (аналог extend)
first_half += second_half
print(first_half)  # [1, 2, 3, 4, 5, 6]
```

Удаление элементов

Python предоставляет несколько способов удаления элементов из списка, различающихся по способу указания удаляемого элемента:

- **Метод `remove()` — удаление по значению**

Метод `remove()` удаляет первое вхождение указанного значения:

```
fruits = ["apple", "banana", "cherry", "banana", "date"]
fruits.remove("banana")  # Удаляется только первый "banana"
print(fruits)  # ['apple', 'cherry', 'banana', 'date']
# Попытка удалить несуществующий элемент вызовет ошибку
# fruits.remove("orange")  # ValueError: list.remove(x): x not in list
```

- **Метод `pop()` — удаление по индексу**

Метод `pop()` удаляет элемент по индексу и возвращает его значение. Без аргументов удаляет последний элемент:

```
numbers = [10, 20, 30, 40, 50]
# Удаление последнего элемента
last = numbers.pop()
print(f"Удален элемент: {last}")  # Удален элемент: 50
print(numbers)  # [10, 20, 30, 40]
# Удаление по индексу
second = numbers.pop(1)  # Удаляем элемент с индексом 1
print(f"Удален элемент: {second}")  # Удален элемент: 20
print(numbers)  # [10, 30, 40]
# pop() часто используется для реализации стека
stack = [1, 2, 3, 4]
while stack:
    item = stack.pop()
    print(f"Обрабатываем: {item}")
```

- **Оператор `del` — удаление по индексу или срезу**

Оператор `del` может удалять элементы по индексу, срезу или весь список:

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Удаление одного элемента
del data[2]  # Удаляем элемент с индексом 2
print(data)  # [1, 2, 4, 5, 6, 7, 8, 9, 10]
```

```
# Удаление среза
del data[2:5]  # Удаляем элементы с индексами 2, 3, 4
print(data)  # [1, 2, 8, 9, 10]
# Удаление с шагом
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[1::2]  # Удаляем каждый второй элемент, начиная с индекса 1
print(numbers)  # [0, 2, 4, 6, 8]
```

- **Метод `clear()` — полная очистка**

Метод `clear()` удаляет все элементы из списка:

```
shopping_list = ["milk", "bread", "eggs", "cheese"]
shopping_list.clear()
print(shopping_list)  # []
print(len(shopping_list))  # 0
```

Поиск элементов

Поиск в списках — одна из базовых операций, которая может выполняться различными способами в зависимости от требований:

- **Оператор `in` — проверка наличия**

Оператор `in` проверяет, содержится ли элемент в списке:

```
available_colors = ["red", "green", "blue", "yellow", "purple"]
user_choice = "blue"
if user_choice in available_colors:
    print(f"Цвет {user_choice} доступен")
else:
    print(f"Цвет {user_choice} недоступен")
# Можно использовать с not
if "orange" not in available_colors:
    print("Оранжевый цвет недоступен")
```

- **Метод `index()` — поиск позиции**

Метод `index()` возвращает индекс первого вхождения элемента:

```
fruits = ["apple", "banana", "cherry", "banana", "date"]
# Поиск индекса элемента
banana_index = fruits.index("banana")
print(f"Первый банан находится на позиции {banana_index}")  # 1
# Поиск в диапазоне
second_banana = fruits.index("banana", banana_index + 1)  # Ищем начиная
со следующей позиции
print(f"Второй банан находится на позиции {second_banana}")  # 3
# Поиск несуществующего элемента вызовет ошибку
# fruits.index("orange")  # ValueError: 'orange' is not in list
# Безопасный поиск
def safe_index(lst, item):
    try:
        return lst.index(item)
```

```
    except ValueError:
        return -1  # Возвращаем -1 если элемент не найден
position = safe_index(fruits, "orange")
print(f"Позиция orange: {position}")  # -1
```

- **Метод `count()` — подсчет вхождений**

Метод `count()` возвращает количество вхождений элемента в список:

```
grades = [5, 4, 5, 3, 5, 4, 2, 5, 4, 3, 5]
excellent_count = grades.count(5)
print(f"Отличных оценок: {excellent_count}")  # 5
good_count = grades.count(4)
print(f"Хороших оценок: {good_count}")  # 3
# Подсчет для всех уникальных элементов
unique_grades = set(grades)  # Получаем уникальные элементы
for grade in sorted(unique_grades):
    count = grades.count(grade)
    print(f"Оценка {grade}: {count} раз(а)")
```

6.3. Срезы: элегантное извлечение подпоследовательностей

Срезы (slices) представляют собой одну из самых элегантных и мощных возможностей Python для работы с последовательностями. Они позволяют извлекать, изменять и копировать части списков (и других последовательностей) с помощью лаконичного и выразительного синтаксиса. Понимание срезов существенно повышает эффективность работы с данными и делает код более читаемым.

Базовый синтаксис срезов

Срез определяется с помощью квадратных скобок и может включать до трех параметров, разделенных двоеточием: `[start:stop:step]`. Каждый из этих параметров необязателен, что создает множество возможностей для гибкой работы с данными:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# Базовые срезы
print(numbers[2:7])    # [2, 3, 4, 5, 6] - с индекса 2 до 7 (не включая 7)
print(numbers[3:])     # [3, 4, 5, 6, 7, 8, 9] - с индекса 3 до конца
print(numbers[:6])     # [0, 1, 2, 3, 4, 5] - с начала до индекса 6
print(numbers[:])      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] - полная копия
```

Важно помнить, что верхняя граница среза не включается в результат. Это соответствует общему принципу Python «включая левую границу, исключая правую», который используется в функции `range()` и других конструкциях.

Работа с отрицательными индексами

Отрицательные индексы в срезах работают так же, как и при обращении к отдельным элементам, позволяя отсчитывать позиции с конца списка:

```
text = "Programming"
chars = list(text)  # ['P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g']
# Срезы с отрицательными индексами
print(chars[-4:])      # ['m', 'i', 'n', 'g'] - последние четыре символа
print(chars[:-3])       # ['P', 'r', 'o', 'g', 'r', 'a', 'm', 'm'] - все,
кроме последних трех
print(chars[-6:-2])    # ['a', 'm', 'm', 'i'] - от -6 до -2 (не включая -2)
# Смешанные положительные и отрицательные индексы
print(chars[2:-2])      # ['o', 'g', 'r', 'a', 'm', 'm', 'i'] - со второго
до предпоследнего
```

Использование шага в срезах

Третий параметр среза определяет шаг — интервал между выбираемыми элементами. Это открывает дополнительные возможности для фильтрации и обработки данных:

```
sequence = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
# Срезы с шагом
print(sequence[::2])    # [0, 2, 4, 6, 8, 10, 12] - каждый второй элемент
print(sequence[1::2])   # [1, 3, 5, 7, 9, 11] - каждый второй, начиная с 1
print(sequence[::3])    # [0, 3, 6, 9, 12] - каждый третий элемент
print(sequence[2:10:2]) # [2, 4, 6, 8] - каждый второй от 2 до 10
# Отрицательный шаг для обращения последовательности
print(sequence[::-1])   # [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] - об-
ращение
print(sequence[::-2])   # [12, 10, 8, 6, 4, 2, 0] - каждый второй в обрат-
ном порядке
print(sequence[8:2:-1]) # [8, 7, 6, 5, 4, 3] - от 8 до 2 в обратном порядке
```

Срезы для копирования и модификации

Срезы создают новые объекты (поверхностные копии), что делает их полезными для копирования данных. Кроме того, срезы можно использовать для изменения частей списков:

```
original = [1, 2, 3, 4, 5]
# Создание копий с помощью срезов
full_copy = original[:]        # Полная копия
partial_copy = original[1:4]   # [2, 3, 4] - частичная копия
# Изменение копии не влияет на оригинал
full_copy[0] = 999
print(original)    # [1, 2, 3, 4, 5] - оригинал не изменился
print(full_copy)   # [999, 2, 3, 4, 5]
# Модификация списка через срезы
numbers = [10, 20, 30, 40, 50, 60]
# Замена части списка
numbers[1:4] = [25, 35, 45]
print(numbers)  # [10, 25, 35, 45, 50, 60]
# Замена на последовательность другой длины
numbers[2:4] = [100, 200, 300, 400]
```

```
print(numbers)  # [10, 25, 100, 200, 300, 400, 50, 60]
# Удаление элементов через присваивание пустого списка
numbers[3:6] = []
print(numbers)  # [10, 25, 100, 50, 60]
```

Срезы строк

Хотя строки неизменяемы, срезы прекрасно работают с ними для извлечения подстрок:

```
url = "https://www.example.com/path/file.html"
# Извлечение частей URL
protocol = url[:8]        # "https://"
domain = url[8:23]        # "www.example.com"
path = url[23:]           # "/path/file.html"
print(f"Протокол: {protocol}")
print(f"Домен: {domain}")
print(f"Путь: {path}")
# Извлечение расширения файла
filename = "document.pdf"
name, extension = filename[:-4], filename[-4:]
print(f"Имя: {name}, Расширение: {extension}")  # Имя: document, Расширение: .pdf
# Обращение строки
text = "Python"
reversed_text = text[::-1]
print(reversed_text)  # "nohtyP"
```

Практические применения срезов

Срезы находят широкое применение в реальных задачах обработки данных.

Обработка данных с фиксированной структурой

```
# Обработка строки с данными фиксированного формата
data_line = "2024102310152545.67USD"
# Извлечение компонентов по позициям
year = data_line[:4]          # "2024"
month = data_line[4:6]        # "10"
day = data_line[6:8]          # "23"
hour = data_line[8:10]        # "10"
minute = data_line[10:12]     # "15"
second = data_line[12:14]     # "25"
amount = data_line[14:20]     # "45.67"
currency = data_line[20:]     # "USD"
print(f"Дата: {day}.{month}.{year}")
print(f"Время: {hour}:{minute}:{second}")
print(f"Сумма: {amount} {currency}")
```

Пагинация данных

```
def paginate_data(data, page_size=5, page_number=1):
```

```
    """Разбивка данных на страницы"""
    start_index = (page_number - 1) * page_size
    end_index = start_index + page_size

    page_data = data[start_index:end_index]
    total_pages = len(data) // page_size + (1 if len(data) % page_size else
0)

    return {
        'data': page_data,
        'page': page_number,
        'total_pages': total_pages,
        'total_items': len(data)
    }
# Пример использования
products = [f"Product {i}" for i in range(1, 23)]  # 22 товара
# Получение первой страницы
page1 = paginate_data(products, page_size=5, page_number=1)
print(f"Страница {page1['page']} из {page1['total_pages']}")
print(f"Товары: {page1['data']}")
# Получение последней страницы
last_page = paginate_data(products, page_size=5, page_number=5)
print(f"\nПоследняя страница: {last_page['data']}")
```

Скользящее окно для анализа временных рядов

```
def moving_window_analysis(data, window_size=3):
    """Анализ данных с помощью скользящего окна"""
    results = []

    for i in range(len(data) - window_size + 1):
        window = data[i:i + window_size]
        window_avg = sum(window) / len(window)
        window_max = max(window)
        window_min = min(window)

        results.append({
            'window': window,
            'average': round(window_avg, 2),
            'maximum': window_max,
            'minimum': window_min
        })

    return results
# Анализ продаж по дням
daily_sales = [150, 200, 175, 300, 250, 180, 220, 190, 280, 160]
analysis = moving_window_analysis(daily_sales, window_size=3)
for i, result in enumerate(analysis):
    print(f"Окно {i+1}: {result['window']} -> "
          f"Среднее: {result['average']}, "
          f"Макс: {result['maximum']}, "
          f"Мин: {result['minimum']}")
```

6.4. Кортежи: неизменяемые последовательности

Кортеж (tuple) — это упорядоченная неизменяемая коллекция элементов. В то время как списки оптимизированы для изменения данных, кортежи предназначены для хранения данных, которые не должны изменяться после создания. Эта неизменяемость (immutability) делает кортежи более быстрыми и безопасными в определенных сценариях использования.

Создание кортежей

Кортежи создаются с помощью круглых скобок, хотя в некоторых случаях скобки можно опустить:

```
# Создание кортежей с круглыми скобками
coordinates = (10, 20)
colors = ("red", "green", "blue")
mixed_tuple = (42, "hello", 3.14, True)
# Создание кортежей без скобок
point = 15, 25
person = "John", "Doe", 30
# Пустой кортеж
empty_tuple = ()
# или
empty_tuple = tuple()
# Кортеж с одним элементом (обратите внимание на запятую!)
single_item = (42,)  # Запятая обязательна!
# Без запятой это просто число в скобках:
not_tuple = (42)     # Это int, а не tuple
print(type(single_item))  # <class 'tuple'>
print(type(not_tuple))    # <class 'int'>
```

Неизменяемость кортежей

Главное отличие кортежей от списков — их неизменяемость. После создания кортежа нельзя изменить его элементы:

```
point = (10, 20)
# point[0] = 15  # TypeError: 'tuple' object does not support item assignment
# Но можно создать новый кортеж
new_point = (15, point[1])  # (15, 20)
# Кортежи можно переназначать (создается новый объект)
coordinates = (1, 2, 3)
coordinates = (4, 5, 6)  # Создается новый кортеж
```

Важно понимать разницу между неизменяемостью самого кортежа и изменяемостью его элементов. Если кортеж содержит изменяемые объекты (например списки), эти объекты можно изменять:

```
# Кортеж содержит изменяемые объекты
data = ([1, 2, 3], [4, 5, 6])
# Нельзя изменить сам кортеж
# data[0] = [7, 8, 9]  # TypeError
```

```
# Но можно изменить содержимое списков внутри кортежа
data[0].append(99)
print(data)  # ([1, 2, 3, 99], [4, 5, 6])
```

Доступ к элементам кортежей

Обращение к элементам кортежей работает точно так же, как для списков:

```
student = ("Alice", "Johnson", 22, "Computer Science")
# Доступ по индексу
name = student[0]        # "Alice"
age = student[2]         # 22
major = student[-1]      # "Computer Science"
# Срезы работают и с кортежами
full_name = student[:2]  # ("Alice", "Johnson")
info = student[1:]       # ("Johnson", 22, "Computer Science")
print(f"Студент: {name} {student[1]}, возраст: {age}")
```

Операции с кортежами

Кортежи поддерживают многие операции, доступные для списков, но только те, которые не изменяют содержимое:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
# Конкатенация создает новый кортеж
combined = tuple1 + tuple2 # (1, 2, 3, 4, 5, 6)
# Умножение кортежа
repeated = (1, 2) * 3      # (1, 2, 1, 2, 1, 2)
# Проверка принадлежности
print(2 in tuple1)         # True
print(7 in tuple1)         # False
# Длина кортежа
print(len(combined))       # 6
# Поиск элементов
numbers = (5, 3, 8, 3, 1, 3, 9)
print(numbers.index(8))    # 2 - индекс первого вхождения
print(numbers.count(3))    # 3 - количество вхождений
```

Распаковка кортежей

Одна из самых мощных возможностей кортежей — распаковка (unpacking), которая позволяет присваивать элементы кортежа нескольким переменным одновременно:

```
# Базовая распаковка
point = (10, 20)
x, y = point
print(f"X: {x}, Y: {y}")  # X: 10, Y: 20
# Распаковка для возврата нескольких значений из функции
def get_name_age():
    return "John", 25
```

```
name, age = get_name_age()
print(f"Имя: {name}, Возраст: {age}")
# Обмен значений переменных
a, b = 1, 2
a, b = b, a  # Элегантный способ обмена значений
print(f"a = {a}, b = {b}")  # a = 2, b = 1
# Распаковка с игнорированием элементов
data = ("John", "Doe", 30, "Engineer", "New York")
first_name, last_name, age, _, city = data  # Профессия игнорируется
print(f"{first_name} {last_name}, {age}, {city}")
```

Распаковка также поддерживает сбор остаточных элементов с помощью оператора *:

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# Сбор элементов в список
first, *middle, last = numbers
print(f"Первый: {first}")    # 1
print(f"Средние: {middle}")  # [2, 3, 4, 5, 6, 7, 8, 9]
print(f"Последний: {last}") # 10
# Распаковка первых нескольких элементов
first, second, *rest = numbers
print(f"Первые два: {first}, {second}")  # 1, 2
print(f"Остальные: {rest}")              # [3, 4, 5, 6, 7, 8, 9, 10]
```

Когда использовать кортежи

Кортежи следует выбирать в следующих ситуациях.

1. Координаты и фиксированные структуры данных

```
# Представление геометрических точек
point_2d = (10, 20)
point_3d = (10, 20, 30)
# RGB цвета
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
# Размеры изображений
image_size = (1920, 1080)  # ширина, высота
```

2. Возврат нескольких значений из функций

```
def calculate_statistics(numbers):
    """Вычисляет статистики для списка чисел"""
    if not numbers:
        return None, None, None

    total = sum(numbers)
    average = total / len(numbers)
    maximum = max(numbers)
    minimum = min(numbers)
```

```
    return total, average, maximum, minimum
# Использование функции
data = [10, 20, 30, 40, 50]
total, avg, max_val, min_val = calculate_statistics(data)
print(f"Сумма: {total}, Среднее: {avg}, Макс: {max_val}, Мин: {min_val}")
```

3. Ключи словарей (благодаря неизменяемости)

```
# Координаты как ключи для карты
game_map = {
    (0, 0): "start",
    (1, 0): "wall",
    (2, 0): "treasure",
    (0, 1): "enemy",
    (1, 1): "empty",
    (2, 1): "exit"
}
# Проверка позиции на карте
player_position = (1, 1)
if player_position in game_map:
    print(f"На позиции {player_position}: {game_map[player_position]}")
```

4. Конфигурационные данные

```
# Настройки подключения к базе данных
DATABASE_CONFIG = (
    "localhost",    # host
    5432,           # port
    "myapp",        # database name
    "username",     # user
    "password"      # password
)
def connect_to_database():
    host, port, db_name, user, password = DATABASE_CONFIG
    print(f"Подключение к {db_name} на {host}:{port}")
    # Логика подключения...
# Список поддерживаемых форматов файлов
SUPPORTED_FORMATS = ("jpg", "png", "gif", "bmp", "tiff")
def is_supported_format(filename):
    extension = filename.split('.')[-1].lower()
    return extension in SUPPORTED_FORMATS
```

Сравнение производительности: список vs кортеж

Кортежи обычно работают быстрее списков для операций чтения благодаря своей неизменяемости:

```
import sys
import time
# Создание больших коллекций
list_data = [i for i in range(1000000)]
tuple_data = tuple(list_data)
```

```
# Сравнение размера в памяти
print(f"Размер списка в памяти: {sys.getsizeof(list_data)} байт")
print(f"Размер кортежа в памяти: {sys.getsizeof(tuple_data)} байт")
# Сравнение времени доступа к элементам
def time_access(collection, iterations=1000000):
    start = time.time()
    for _ in range(iterations):
        _ = collection[500000]  # Доступ к элементу в середине
    return time.time() - start
list_time = time_access(list_data)
tuple_time = time_access(tuple_data)
print(f"Время доступа к элементу списка: {list_time:.4f} сек")
print(f"Время доступа к элементу кортежа: {tuple_time:.4f} сек")
```

Именованные кортежи

Python предоставляет дополнительную возможность — именованные кортежи (named tuples), которые сочетают эффективность кортежей с удобством доступа к полям по именам:

```
from collections import namedtuple
# Создание именованного кортежа
Point = namedtuple('Point', ['x', 'y'])
Student = namedtuple('Student', ['name', 'age', 'grade'])
# Использование именованных кортежей
p1 = Point(10, 20)
print(f"Координаты: x={p1.x}, y={p1.y}")
student = Student("Alice", 20, "A")
print(f"Студент: {student.name}, возраст: {student.age}, оценка: {student.
grade}")
# Именованные кортежи остаются кортежами
print(student[0])  # "Alice" - доступ по индексу тоже работает
name, age, grade = student  # Распаковка работает
```

Кортежи представляют собой важную структуру данных для случаев, когда нужна упорядоченная коллекция неизменяемых элементов. Их неизменяемость обеспечивает безопасность данных и позволяет использовать их в качестве ключей словарей, а также делает код более предсказуемым и менее подверженным ошибкам.

6.5. Словари: ключ-значение в действии

Словарь (`dictionary` или `dict`) — это неупорядоченная изменяемая коллекция пар «ключ-значение», где каждый ключ уникален и служит для быстрого доступа к соответствующему значению. Словари представляют собой одну из самых мощных и универсальных структур данных в Python, идеально подходящую для моделирования связей между объектами реального мира.

В отличие от списков и кортежей, где доступ к элементам осуществляется по числовому индексу, словари используют ключи произвольного типа (с некоторыми ограничениями). Это делает их незаменимыми для решения задач, где данные име-

ют естественные идентификаторы: телефонные справочники, каталоги товаров, пользовательские профили и многое другое.

Создание словарей

Словари создаются с помощью фигурных скобок, где каждая пара «ключ-значение» разделяется двоеточием:

```
# Пустой словарь
empty_dict = {}
# или
empty_dict = dict()
# Словарь с данными
student = {
    "name": "Alice Johnson",
    "age": 20,
    "major": "Computer Science",
    "gpa": 3.85
}
# Словарь с различными типами ключей
mixed_keys = {
    "text_key": "string value",
    42: "numeric key",
    (1, 2): "tuple key",
    True: "boolean key"
}
# Создание словаря с помощью функции dict()
person = dict(name="John", surname="Doe", age=30)
# Эквивалентно: person = {"name": "John", "surname": "Doe", "age": 30}
# Создание словаря из списка кортежей
pairs = [("apple", 5), ("banana", 3), ("orange", 8)]
fruit_count = dict(pairs)  # {"apple": 5, "banana": 3, "orange": 8}
```

Доступ к значениям

Python предоставляет несколько способов получения значений из словаря, каждый из которых подходит для разных ситуаций:

```
student = {
    "name": "Alice Johnson",
    "age": 20,
    "major": "Computer Science",
    "courses": ["Python", "Algorithms", "Databases"]
}
# Доступ через квадратные скобки
name = student["name"]        # "Alice Johnson"
age = student["age"]          # 20
# Попытка доступа к несуществующему ключу вызовет ошибку
```

```
# grade = student["grade"]    # KeyError: 'grade'
# Безопасный доступ с помощью метода get()
grade = student.get("grade")          # None (по умолчанию)
grade = student.get("grade", "N/A")    # "N/A" (заданное значение по умол-
чанию)
gpa = student.get("gpa", 0.0)          # 0.0
# Проверка наличия ключа
if "email" in student:
    print(f"Email: {student['email']}")
else:
    print("Email не указан")
# Альтернативная проверка
has_phone = "phone" in student  # False
```

Изменение словарей

Словари являются изменяемыми, что позволяет добавлять, обновлять и удалять пары «ключ-значение»:

```
inventory = {
    "apples": 50,
    "bananas": 30,
    "oranges": 25
}
# Добавление нового элемента
inventory["pears"] = 15
print(inventory)  # {"apples": 50, "bananas": 30, "oranges": 25, "pears":
15}
# Обновление существующего значения
inventory["apples"] = 45
print(inventory["apples"])  # 45
# Обновление нескольких значений с помощью update()
inventory.update({"bananas": 35, "grapes": 20})
print(inventory)  # Обновлены bananas и добавлен grapes
# Обновление из другого словаря
new_items = {"kiwi": 12, "mango": 8}
inventory.update(new_items)
# Удаление элементов
del inventory["oranges"]  # Удаление по ключу
# Удаление с получением значения
removed_value = inventory.pop("pears", 0)  # Вернет значение или 0, если
ключа нет
print(f"Удалено груш: {removed_value}")
# Удаление произвольного элемента (в версиях Python 3.7+ - последнего)
if inventory:
    key, value = inventory.popitem()
    print(f"Удален элемент: {key} = {value}")
# Очистка словаря
# inventory.clear()  # Удаляет все элементы
```

Методы работы со словарями

Словари предоставляют богатый набор методов для работы с ключами, значениями и парами:

```
grades = {
    "Alice": 85,
    "Bob": 92,
    "Charlie": 78,
    "Diana": 96,
    "Eve": 88
}
# Получение всех ключей, значений и пар
keys = grades.keys()      # dict_keys(['Alice', 'Bob', 'Charlie', 'Diana',
'Eve'])
values = grades.values()  # dict_values([85, 92, 78, 96, 88])
items = grades.items()    # dict_items([('Alice', 85), ('Bob', 92), ...])
# Преобразование в списки (если нужно)
student_names = list(grades.keys())
all_grades = list(grades.values())
# Итерация по словарю
print("Оценки студентов:")
for name, grade in grades.items():
    print(f"{name}: {grade}")
print("\nСписок студентов:")
for name in grades:  # По умолчанию итерация происходит по ключам
    print(f"- {name}")
print("\nВсе оценки:")
for grade in grades.values():
    print(f"Оценка: {grade}")
```

Вложенные словари

Словари могут содержать другие словари, создавая сложные иерархические структуры данных:

```
company = {
    "name": "TechCorp",
    "employees": {
        "101": {
            "name": "John Smith",
            "position": "Developer",
            "salary": 75000,
            "skills": ["Python", "JavaScript", "SQL"]
        },
        "102": {
            "name": "Alice Johnson",
            "position": "Designer",
            "salary": 65000,
            "skills": ["Photoshop", "Figma", "CSS"]
        },
```

```
        "103": {
            "name": "Bob Wilson",
            "position": "Manager",
            "salary": 85000,
            "skills": ["Leadership", "Planning", "Communication"]
        }
    },
    "departments": {
        "development": ["101"],
        "design": ["102"],
        "management": ["103"]
    }
}
# Доступ к вложенным данным
john_salary = company["employees"]["101"]["salary"]
print(f"Зарплата John Smith: ${john_salary}")
# Получение всех навыков в компании
all_skills = set()
for emp_id, emp_data in company["employees"].items():
    all_skills.update(emp_data["skills"])
print(f"Все навыки в компании: {sorted(all_skills)}")
# Безопасный доступ к глубоко вложенным данным
def get_nested_value(dictionary, keys, default=None):
    """Безопасное получение значения из вложенного словаря"""
    for key in keys:
        if isinstance(dictionary, dict) and key in dictionary:
            dictionary = dictionary[key]
        else:
            return default
    return dictionary
# Использование функции
phone = get_nested_value(company, ["employees", "101", "phone"], "Не ука-
зан")
print(f"Телефон John Smith: {phone}")
```

6.6. Множества: уникальность и математические операции

Множества (sets) представляют собой одну из самых элегантных структур данных в Python, воплощающую математическую концепцию множества в программировании. Основная особенность множеств — они содержат только уникальные элементы и не имеют определенного порядка. Эта структура данных становится незаменимой, когда необходимо работать с уникальными значениями или выполнять операции, характерные для математической теории множеств.

Концептуально множество можно представить как контейнер, который автоматически отбрасывает дубликаты. Если список подобен записной книжке, где могут быть повторяющиеся записи, а словарь напоминает телефонный справочник с уникальными ключами, то множество похоже на коллекцию уникальных марок — каждая марка может присутствовать только в одном экземпляре.

Создание множеств

Множества в Python создаются несколькими способами, каждый из которых подходит для определенных ситуаций:

```
# Создание множества с помощью фигурных скобок
colors = {'red', 'green', 'blue', 'red', 'yellow'}
print(colors)  # {'blue', 'green', 'red', 'yellow'} - дубликат 'red' удален
# Создание пустого множества (внимание: {} создает словарь, не множество!)
empty_set = set()
print(type(empty_set))  # <class 'set'>
# Создание множества из списка (автоматическое удаление дубликатов)
numbers = set([1, 2, 3, 2, 4, 3, 5])
print(numbers)  # {1, 2, 3, 4, 5}
# Создание множества из строки
letters = set("programming")
print(letters)  # {'a', 'g', 'i', 'm', 'n', 'o', 'p', 'r'}
```

Важная деталь: пустое множество нельзя создать с помощью {}, поскольку такая запись создает пустой словарь. Это одна из немногих несогласованностей в синтаксисе Python, которую важно запомнить.

Операции с элементами множества

Множества предоставляют интуитивно понятные методы для добавления, удаления и проверки присутствия элементов:

```
# Создаем множество для демонстрации операций
fruits = {'apple', 'banana', 'orange'}
# Добавление элемента
fruits.add('grape')
print(fruits)  # {'grape', 'banana', 'apple', 'orange'}
# Попытка добавить существующий элемент (ничего не произойдет)
fruits.add('apple')
print(len(fruits))  # Размер не изменился
# Добавление нескольких элементов
fruits.update(['kiwi', 'mango', 'apple'])  # 'apple' уже есть, не добавится
print(fruits)
# Удаление элемента (с ошибкой, если элемента нет)
fruits.remove('banana')
print(fruits)
# Безопасное удаление (без ошибки, если элемента нет)
fruits.discard('banana')  # Ошибки не будет, даже если 'banana' уже удален
fruits.discard('nonexistent')  # Ошибки тоже не будет
# Проверка принадлежности
if 'apple' in fruits:
    print("Apple is in the set")
# Получение и удаление случайного элемента
random_fruit = fruits.pop()
print(f"Removed: {random_fruit}")
print(f"Remaining: {fruits}")
```

Математические операции с множествами

Истинная сила множеств раскрывается в математических операциях, которые позволяют элегантно решать задачи на пересечения, объединения и различия:

```
# Создаем несколько множеств для демонстрации
python_developers = {'alice', 'bob', 'charlie', 'diana'}
java_developers = {'bob', 'charlie', 'eve', 'frank'}
mobile_developers = {'charlie', 'diana', 'eve', 'george'}
# Объединение (union) - все уникальные элементы
all_developers = python_developers | java_developers | mobile_developers
print(f"All developers: {all_developers}")
# Альтернативный синтаксис объединения
all_devs_method = python_developers.union(java_developers, mobile_
developers)
print(f"Same result: {all_devs_method}")
# Пересечение (intersection) - общие элементы
multilingual_devs = python_developers & java_developers
print(f"Python and Java developers: {multilingual_devs}")
# Пересечение трех множеств
versatile_devs = python_developers & java_developers & mobile_developers
print(f"Developers knowing all three: {versatile_devs}")
# Разность (difference) - элементы из первого, но не из второго
python_only = python_developers - java_developers
print(f"Only Python developers: {python_only}")
# Симметрическая разность (symmetric difference) - элементы, которые есть
# в одном из множеств, но не в обоих
either_python_or_java = python_developers ^ java_developers
print(f"Either Python or Java (not both): {either_python_or_java}")
```

Эти операции имеют глубокий практический смысл. Представьте систему управления доступом к корпоративным ресурсам — множества позволяют элегантно находить пользователей с определенными комбинациями прав доступа.

Практические применения множеств

Рассмотрим несколько реальных сценариев, где множества становятся незаменимыми:

```
# Пример 1: Анализ уникальных посетителей веб-сайта
def analyze_website_traffic():
    # Данные о посещениях по дням (ID пользователей)
    monday_visitors = {101, 102, 103, 104, 105, 101, 102}  # дубликаты ав-
томатически удалятся
    tuesday_visitors = {102, 104, 106, 107, 108}
    wednesday_visitors = {103, 105, 107, 109, 110}

    # Уникальные посетители за неделю
   unique_visitors = monday_visitors | tuesday_visitors | wednesday_visitors
    print(f"Total unique visitors: {len(unique_visitors)}")
```

```
    # Постоянные посетители (были все три дня)
      regular_visitors = monday_visitors & tuesday_visitors & wednesday_
visitors
    print(f"Regular visitors: {regular_visitors}")

    # Посетители только в понедельник
    monday_only = monday_visitors - tuesday_visitors - wednesday_visitors
    print(f"Monday-only visitors: {monday_only}")
analyze_website_traffic()
# Пример 2: Проверка прав доступа в системе
def check_access_rights():
    # Определяем группы пользователей
    admin_users = {'admin', 'root', 'supervisor'}
    read_users = {'user1', 'user2', 'user3', 'admin', 'guest'}
    write_users = {'user1', 'admin', 'editor', 'supervisor'}

    # Функция для проверки доступа
    def can_user_write(username):
        return username in write_users

    def can_user_read(username):
        return username in read_users

    def is_admin(username):
        return username in admin_users

    # Анализ пользователей
    all_users = admin_users | read_users | write_users
    print(f"All users in system: {all_users}")

    # Пользователи с правами чтения, но без прав записи
    read_only_users = read_users - write_users
    print(f"Read-only users: {read_only_users}")

    # Администраторы с правами записи
    admin_writers = admin_users & write_users
    print(f"Admin writers: {admin_writers}")
check_access_rights()
```

Множества и производительность

Одно из ключевых преимуществ множеств — исключительная скорость операций поиска. Проверка принадлежности элемента множеству выполняется в среднем за константное время `O(1)`, в отличие от списков, где требуется `O(n)` времени:

```
import time
# Создаем большие коллекции для тестирования
large_list = list(range(100000))
large_set = set(range(100000))
search_element = 99999  # Элемент в конце
```

```
# Тестируем скорость поиска в списке
start_time = time.time()
for _ in range(1000):
    result = search_element in large_list
list_time = time.time() - start_time
# Тестируем скорость поиска в множестве
start_time = time.time()
for _ in range(1000):
    result = search_element in large_set
set_time = time.time() - start_time
print(f"List search time: {list_time:.4f} seconds")
print(f"Set search time: {set_time:.4f} seconds")
print(f"Set is {list_time/set_time:.1f} times faster")
```

Неизменяемые множества (`frozenset`)

Python также предоставляет неизменяемую версию множеств — `frozenset`. Эта структура данных полезна, когда нужно создать множество, которое нельзя изменить после создания, или использовать множество в качестве ключа словаря:

```
# Создание неизменяемого множества
immutable_colors = frozenset(['red', 'green', 'blue'])
print(immutable_colors)
# frozenset можно использовать как ключ словаря
color_combinations = {
    frozenset(['red', 'blue']): 'purple',
    frozenset(['red', 'yellow']): 'orange',
    frozenset(['blue', 'yellow']): 'green'
}
# Поиск комбинации цветов
search_combo = frozenset(['red', 'blue'])
if search_combo in color_combinations:
     print(f"Combination {search_combo} makes {color_combinations[search_
combo]}")
```

Практические рекомендации по использованию множеств

При работе с множествами следует помнить несколько важных принципов. Используйте множества, когда порядок элементов не важен, но критична уникальность. Множества идеально подходят для удаления дубликатов из коллекций, выполнения математических операций над наборами данных и быстрой проверки принадлежности элементов.

Избегайте множеств, если необходимо сохранить порядок элементов или если элементы коллекции должны быть изменяемыми объектами. Множества могут содержать только неизменяемые (хешируемые) объекты — числа, строки, кортежи, но не списки или словари.

```
# Пример элегантного удаления дубликатов с сохранением типа данных
def remove_duplicates_preserve_order(items):
    """Удаляет дубликаты, сохраняя порядок первого появления элемента"""
    seen = set()
```

```
    result = []
    for item in items:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result
# Тестирование функции
original_list = ['apple', 'banana', 'apple', 'cherry', 'banana', 'date']
cleaned_list = remove_duplicates_preserve_order(original_list)
print(f"Original: {original_list}")
print(f"Cleaned: {cleaned_list}")
```

Множества в Python представляют собой мощный инструмент для работы с уникальными данными и выполнения операций, вдохновленных математической теорией множеств. Их эффективность в операциях поиска и элегантность в решении задач на пересечения и объединения делают множества незаменимыми в арсенале Python-разработчика. Понимание, когда и как использовать множества — важный шаг к написанию эффективного и выразительного кода.

6.7. Выбор правильной структуры данных

Выбор подходящей структуры данных — одно из важнейших решений в программировании, которое определяет не только производительность программы, но и читаемость кода. Каждая структура данных в Python оптимизирована для определенных операций и сценариев использования. Понимание сильных и слабых сторон каждой структуры позволяет писать более эффективный и элегантный код.

Принятие решения о выборе структуры данных должно основываться на анализе нескольких ключевых факторов: типе операций, которые будут выполняться наиболее часто, необходимости сохранения порядка элементов, требованиях к уникальности данных, потребности в изменяемости структуры и ожидаемом объеме данных.

Критерии выбора структуры данных

Первый критерий — **частота различных операций**. Если программа будет часто выполнять поиск элементов, множества и словари предоставляют значительные преимущества благодаря константному времени доступа. Для частых операций добавления и удаления в конце коллекции идеально подходят списки. Если основная операция — перебор элементов без изменений, кортежи могут быть оптимальным выбором.

```
# Демонстрация влияния выбора структуры на производительность
import time
def performance_comparison():
    # Создаем данные для тестирования
    data = list(range(10000))

    # Тест 1: Поиск элемента
    search_list = data.copy()
    search_set = set(data)
```

```
    search_dict = {i: f"value_{i}" for i in data}

    target = 9999

    # Поиск в списке
    start = time.time()
    for _ in range(1000):
        result = target in search_list
    list_search_time = time.time() - start

    # Поиск в множестве
    start = time.time()
    for _ in range(1000):
        result = target in search_set
    set_search_time = time.time() - start

    # Поиск в словаре
    start = time.time()
    for _ in range(1000):
        result = target in search_dict
    dict_search_time = time.time() - start

    print("Performance comparison for search operations:")
    print(f"List search: {list_search_time:.6f} seconds")
    print(f"Set search: {set_search_time:.6f} seconds")
    print(f"Dict search: {dict_search_time:.6f} seconds")
performance_comparison()
```

Второй критерий — **необходимость сохранения порядка**. Списки и кортежи сохраняют порядок вставки элементов, что критично для многих алгоритмов. Словари в Python 3.7+ также сохраняют порядок вставки, но это считается деталью реализации. Множества не гарантируют сохранения порядка.

Третий критерий — **требования к уникальности**. Если данные должны быть уникальными, множества автоматически обеспечивают это свойство. Словари гарантируют уникальность ключей, но могут иметь повторяющиеся значения.

Практические сценарии выбора

Рассмотрим типичные задачи программирования и оптимальные структуры данных для их решения:

```
# Сценарий 1: Система счетчиков и статистики
def choose_for_counting():
    # Плохой выбор: список кортежей
    bad_counters = [('apples', 5), ('bananas', 3), ('apples', 2)]

    def count_bad(item):
        total = 0
        for name, count in bad_counters:
            if name == item:
                total += count
        return total
```

```
        # Хороший выбор: словарь
    good_counters = {'apples': 0, 'bananas': 0, 'oranges': 0}

    def count_good(item, amount):
        good_counters[item] = good_counters.get(item, 0) + amount

    # Демонстрация разницы
    print("Bad approach:", count_bad('apples'))  # Медленный поиск

    count_good('apples', 5)
    count_good('apples', 2)
    print("Good approach:", good_counters['apples'])  # Быстрый доступ
choose_for_counting()
# Сценарий 2: Обработка последовательности действий
def choose_for_sequence():
    # Хороший выбор для последовательности: список
    user_actions = []

    def add_action(action):
        user_actions.append(action)
        # Ограничиваем историю последними 100 действиями
        if len(user_actions) > 100:
            user_actions.pop(0)

    def get_recent_actions(count=5):
        return user_actions[-count:]

    def undo_last_action():
        if user_actions:
            return user_actions.pop()
        return None

    # Демонстрация использования
    add_action("open_file")
    add_action("edit_text")
    add_action("save_file")

    print("Recent actions:", get_recent_actions())
    undone = undo_last_action()
    print("Undone action:", undone)
    print("Actions after undo:", user_actions)
choose_for_sequence()
# Сценарий 3: Конфигурационные данные
def choose_for_config():
    # Хороший выбор: кортеж для неизменяемых настроек
    DATABASE_CONFIG = (
```

```
        'localhost',
        5432,
        'myapp_db',
        'production'
    )

    # Плохо: список (может быть случайно изменен)
    # DATABASE_CONFIG = ['localhost', 5432, 'myapp_db', 'production']

    # Хороший выбор: словарь для именованных настроек
    APP_SETTINGS = {
        'debug': False,
        'max_connections': 100,
        'timeout': 30,
         'allowed_hosts': {'example.com', 'api.example.com'}  # множество
для уникальных хостов
    }

    def is_host_allowed(host):
        return host in APP_SETTINGS['allowed_hosts']

    # Демонстрация безопасности кортежа
    try:
        DATABASE_CONFIG[0] = 'newhost'  # Ошибка! Кортеж неизменяем
    except TypeError as e:
        print("Configuration protected:", str(e))

    print("Host check:", is_host_allowed('example.com'))
choose_for_config()
```

Практические рекомендации

При выборе структуры данных следуйте принципу «измеряй, не предполагай». Начинайте с самой простой структуры, которая решает задачу, и оптимизируйте только при необходимости. Списки универсальны и подходят для большинства задач, где не критична скорость поиска. Словари идеальны для создания индексов и быстрого доступа по ключу. Множества незаменимы для работы с уникальными элементами и выполнения операций пересечения или объединения. Кортежи следует использовать для неизменяемых данных и в качестве ключей словарей.

Помните, что преждевременная оптимизация — корень всех зол в программировании. Сначала напишите работающий код с понятной логикой, затем профилируйте узкие места и оптимизируйте только там, где это действительно необходимо. Часто читаемость кода важнее микрооптимизаций производительности.

Выбор правильной структуры данных — это навык, который развивается с опытом. Понимание компромиссов между различными структурами и умение анализировать требования задачи позволяет создавать эффективные и элегантные решения. В следующих разделах мы изучим, как комбинировать различные структуры для создания более сложных и мощных конструкций данных.

6.8. Вложенные структуры: списки словарей и словари списков

Реальные данные редко имеют простую плоскую структуру. Чаще всего информация организована иерархически, с множественными связями и сложными отношениями между элементами. Именно здесь раскрывается истинная мощь Python — возможность создавать сложные структуры данных, комбинируя простые типы в элегантные и эффективные решения.

Вложенные структуры данных представляют собой композицию различных коллекций, где элементами одной структуры являются другие структуры. Это позволяет моделировать сложные предметные области: от простых каталогов товаров до сложных иерархий организаций или многомерных научных данных.

Понимание принципов работы с вложенными структурами критически важно для любого Python-разработчика, поскольку такие структуры повсеместно встречаются в веб-разработке (JSON-данные), анализе данных, конфигурационных файлах и API различных сервисов.

Списки словарей: коллекции записей

Список словарей — одна из наиболее распространенных вложенных структур, которая естественным образом моделирует таблицы данных или коллекции записей с одинаковой структурой. Каждый словарь в списке представляет отдельную запись, а ключи словарей служат именами полей.

```
# Классический пример: база данных студентов
students = [
    {
        'name': 'Alice Johnson',
        'age': 20,
        'major': 'Computer Science',
        'grades': [85, 92, 78, 94],
        'active': True
    },
    {
        'name': 'Bob Smith',
        'age': 19,
        'major': 'Mathematics',
        'grades': [88, 76, 91, 83],
        'active': True
    },
    {
        'name': 'Charlie Brown',
        'age': 21,
        'major': 'Physics',
        'grades': [92, 89, 95, 87],
        'active': False
    }
]
# Обработка данных: поиск студентов по критериям
def find_students_by_major(students, major):
```

```
    """Найти всех студентов определенной специальности"""
    return [student for student in students if student['major'] == major]
def calculate_average_grade(student):
    """Вычислить средний балл студента"""
    grades = student['grades']
    return sum(grades) / len(grades) if grades else 0
def get_top_students(students, min_average=90):
    """Найти студентов с высоким средним баллом"""
    top_students = []
    for student in students:
        avg_grade = calculate_average_grade(student)
        if avg_grade >= min_average:
            top_students.append({
                'name': student['name'],
                'major': student['major'],
                'average': round(avg_grade, 2)
            })
    return top_students
# Демонстрация использования
cs_students = find_students_by_major(students, 'Computer Science')
print("Computer Science students:")
for student in cs_students:
    print(f"  {student['name']}, age {student['age']}")
top_performers = get_top_students(students, 90)
print("\nTop performing students:")
for student in top_performers:
   print(f"  {student['name']} ({student['major']}): {student['average']}")
```

Словари списков: группировка данных

Словарь списков представляет обратную структуру, где каждый ключ связан со списком значений. Эта структура идеально подходит для группировки данных по категориям или создания индексов.

```
# Пример: система управления задачами
task_system = {
    'todo': [
        {'id': 1, 'task': 'Review code', 'priority': 'high'},
        {'id': 2, 'task': 'Update documentation', 'priority': 'medium'},
        {'id': 3, 'task': 'Fix login bug', 'priority': 'critical'}
    ],
    'in_progress': [
        {'id': 4, 'task': 'Implement new feature', 'priority': 'high'},
        {'id': 5, 'task': 'Database optimization', 'priority': 'medium'}
    ],
    'done': [
        {'id': 6, 'task': 'Setup CI/CD pipeline', 'priority': 'high'},
        {'id': 7, 'task': 'Team meeting', 'priority': 'low'}
    ]
}
```

```
def add_task(system, status, task_data):
    """Добавить новую задачу в систему"""
    if status not in system:
        system[status] = []
    system[status].append(task_data)
def move_task(system, task_id, from_status, to_status):
    """Переместить задачу между статусами"""
    # Найти и удалить задачу из исходного статуса
    task = None
    for i, t in enumerate(system[from_status]):
        if t['id'] == task_id:
            task = system[from_status].pop(i)
            break

    # Добавить в новый статус
    if task:
        if to_status not in system:
            system[to_status] = []
        system[to_status].append(task)
        return True
    return False
def get_tasks_by_priority(system, priority):
    """Получить все задачи определенного приоритета"""
    result = []
    for status, tasks in system.items():
        for task in tasks:
            if task['priority'] == priority:
                result.append({
                    'status': status,
                    'task': task['task'],
                    'id': task['id']
                })
    return result
# Демонстрация работы системы
print("Current system state:")
for status, tasks in task_system.items():
    print(f"{status}: {len(tasks)} tasks")
# Перемещение задачи
move_task(task_system, 1, 'todo', 'in_progress')
print("\nAfter moving task 1 to 'in_progress':")
print(f"todo: {len(task_system['todo'])} tasks")
print(f"in_progress: {len(task_system['in_progress'])} tasks")
# Поиск критических задач
critical_tasks = get_tasks_by_priority(task_system, 'critical')
print(f"\nCritical tasks: {len(critical_tasks)}")
for task in critical_tasks:
    print(f"  {task['task']} (Status: {task['status']})")
```

Рекомендации по дизайну вложенных структур

При проектировании сложных вложенных структур следуйте нескольким важным принципам. Сохраняйте согласованность структуры — если список содержит словари, все словари должны иметь одинаковый набор ключей или четко определенную схему. Избегайте чрезмерной вложенности — более трех-четырех уровней могут сделать код трудночитаемым и подверженным ошибкам.

Предусматривайте обработку отсутствующих данных и используйте значения по умолчанию. Документируйте структуру данных, особенно для сложных иерархий. Рассмотрите возможность использования классов для инкапсуляции логики работы со сложными структурами.

```
# Пример хорошо структурированного дизайна
def create_product_catalog():
    """Создать хорошо структурированный каталог товаров"""
    return {
        'catalog_version': '1.0',
        'last_updated': '2024-03-15',
        'categories': {
            'electronics': {
                'name': 'Electronics',
                'description': 'Electronic devices and accessories',
                'products': [
                    {
                        'id': 'ELEC001',
                        'name': 'Smartphone XR',
                        'price': 699.99,
                        'in_stock': True,
                        'specifications': {
                            'screen_size': '6.1 inch',
                            'storage': '128GB',
                            'camera': '12MP'
                        },
                        'reviews': [
                            {'rating': 5, 'comment': 'Excellent phone!'},
                           {'rating': 4, 'comment': 'Good value for money'}
                        ]
                    }
                ]
            },
            'books': {
                'name': 'Books',
                'description': 'Physical and digital books',
                'products': [
                    {
                        'id': 'BOOK001',
                        'name': 'Python Programming Guide',
                        'price': 39.99,
                        'in_stock': True,
                        'specifications': {
                            'pages': 450,
```

```
                        'author': 'Jane Smith',
                        'format': 'hardcover'
                    },
                    'reviews': []
                }
            ]
        }
    }
}
# Функции для работы с каталогом
def search_products(catalog, search_term):
    """Поиск товаров по названию"""
    results = []
    for category_data in catalog['categories'].values():
        for product in category_data['products']:
            if search_term.lower() in product['name'].lower():
                results.append({
                    'category': category_data['name'],
                    'product': product['name'],
                    'price': product['price'],
                    'id': product['id']
                })
    return results
def get_products_in_price_range(catalog, min_price, max_price):
    """Найти товары в заданном ценовом диапазоне"""
    results = []
    for category_data in catalog['categories'].values():
        for product in category_data['products']:
            if min_price <= product['price'] <= max_price:
                results.append(product)
    return results
# Демонстрация работы
catalog = create_product_catalog()
phone_results = search_products(catalog, 'phone')
affordable_products = get_products_in_price_range(catalog, 0, 50)
print("Phone search results:", len(phone_results))
print("Products under $50:", len(affordable_products))
```

Вложенные структуры данных — мощный инструмент для моделирования сложных предметных областей. Умение правильно проектировать, навигировать и обрабатывать такие структуры является ключевым навыком Python-разработчика. Эти знания особенно важны при работе с API, базами данных, конфигурационными файлами и любыми системами, где данные имеют иерархическую природу.

В следующем разделе мы применим все изученные концепции структур данных для создания полноценного практического проекта — системы учета студентов, которая продемонстрирует, как различные структуры данных работают вместе для решения реальных задач.

6.9. Практический проект: система учета студентов

Завершим изучение структур данных созданием полноценной системы управления студентами, которая объединит все изученные концепции в реальном практическом проекте. Эта система продемонстрирует, как правильный выбор и комбинирование структур данных позволяют создавать эффективные и масштабируемые решения.

Система учета студентов — классический пример задачи, требующей работы с иерархическими данными, поиска, фильтрации и агрегации информации. Мы создадим систему, которая будет управлять студентами, курсами, оценками и расписанием, используя оптимальные структуры данных для каждой задачи.

Архитектура системы и выбор структур данных

Прежде чем приступить к реализации, проанализируем требования и выберем подходящие структуры данных для каждого компонента системы:

```
"""
Система учета студентов - комплексный проект
Демонстрирует использование всех основных структур данных Python
"""
import datetime
from typing import Dict, List, Set, Tuple, Optional
# Глобальная структура данных системы
class StudentManagementSystem:
    def __init__(self):
        # Основные данные студентов (список словарей)
        self.students = []

        # Индекс студентов по ID для быстрого поиска (словарь)
        self.student_index = {}

        # Курсы и их информация (словарь)
        self.courses = {}

        # Регистрация студентов на курсы (словарь множеств)
        self.course_enrollments = {}

        # Оценки студентов (вложенная структура)
        self.grades = {}

        # Расписание занятий (список кортежей)
        self.schedule = []

        # Счетчик для генерации уникальных ID
        self.next_student_id = 1
        self.next_course_id = 1
    def generate_student_id(self):
        """Генерация уникального ID студента"""
        student_id = f"STU{self.next_student_id:04d}"
        self.next_student_id += 1
```

```
        return student_id
    def generate_course_id(self):
        """Генерация уникального ID курса"""
        course_id = f"CRS{self.next_course_id:04d}"
        self.next_course_id += 1
        return course_id
```

Управление студентами

Начнем с базовой функциональности управления студентами, демонстрируя работу со списками словарей и индексирование для оптимизации поиска:

```
def add_student(self, name: str, email: str, birth_date: str, major: str):
        """Добавить нового студента в систему"""
        student_id = self.generate_student_id()

        # Создаем запись студента (словарь с полной информацией)
        student = {
            'id': student_id,
            'name': name,
            'email': email,
            'birth_date': birth_date,
            'major': major,
            'enrollment_date': datetime.date.today().isoformat(),
            'active': True,
            'gpa': 0.0
        }

        # Добавляем в основной список
        self.students.append(student)

        # Создаем индекс для быстрого поиска
        self.student_index[student_id] = len(self.students) - 1

        # Инициализируем структуры для оценок
        self.grades[student_id] = {}

        print(f"Student {name} added with ID: {student_id}")
        return student_id
    def find_student(self, student_id: str) -> Optional[Dict]:
        """Найти студента по ID (быстрый поиск через индекс)"""
        if student_id in self.student_index:
            index = self.student_index[student_id]
            return self.students[index]
        return None
    def search_students_by_name(self, search_term: str) -> List[Dict]:
        """Поиск студентов по имени (линейный поиск)"""
        results = []
        search_lower = search_term.lower()
```

```
        for student in self.students:
            if search_lower in student['name'].lower():
                results.append(student)

        return results
    def get_students_by_major(self, major: str) -> List[Dict]:
        """Получить всех студентов определенной специальности"""
        return [student for student in self.students
                        if student['major'].lower() == major.lower() and
student['active']]
    def update_student_info(self, student_id: str, **updates):
        """Обновить информацию о студенте"""
        student = self.find_student(student_id)
        if student:
            for key, value in updates.items():
                if key in student:
                    student[key] = value
            print(f"Student {student_id} updated successfully")
            return True
        else:
            print(f"Student {student_id} not found")
            return False
```

Управление курсами и регистрацией

Система курсов демонстрирует использование словарей для хранения курсов и множеств для управления регистрацией студентов:

```
def add_course(self, name: str, instructor: str, credits: int,
                   max_students: int = 30) -> str:
        """Добавить новый курс"""
        course_id = self.generate_course_id()

        # Информация о курсе (словарь)
        course = {
            'id': course_id,
            'name': name,
            'instructor': instructor,
            'credits': credits,
            'max_students': max_students,
            'active': True
        }

        # Сохраняем курс
        self.courses[course_id] = course

        # Инициализируем множество зарегистрированных студентов
        self.course_enrollments[course_id] = set()

        print(f"Course '{name}' added with ID: {course_id}")
        return course_id
```

```
    def enroll_student(self, student_id: str, course_id: str) -> bool:
        """Зарегистрировать студента на курс"""
        # Проверяем существование студента и курса
        if not self.find_student(student_id):
            print(f"Student {student_id} not found")
            return False

        if course_id not in self.courses:
            print(f"Course {course_id} not found")
            return False

        # Проверяем лимит студентов
        course = self.courses[course_id]
        current_enrollment = len(self.course_enrollments[course_id])

        if current_enrollment >= course['max_students']:
            print(f"Course {course_id} is full")
            return False

       # Регистрируем студента (множество автоматически исключает дубликаты)
        self.course_enrollments[course_id].add(student_id)

        # Инициализируем оценки для студента по курсу
        if student_id not in self.grades:
            self.grades[student_id] = {}
        self.grades[student_id][course_id] = []

        print(f"Student {student_id} enrolled in course {course_id}")
        return True
    def get_course_students(self, course_id: str) -> List[Dict]:
        """Получить список студентов курса"""
        if course_id not in self.course_enrollments:
            return []

        student_ids = self.course_enrollments[course_id]
        students = []

        for student_id in student_ids:
            student = self.find_student(student_id)
            if student:
                students.append(student)

        return students
    def get_student_courses(self, student_id: str) -> List[Dict]:
        """Получить список курсов студента"""
        courses = []

       for course_id, enrolled_students in self.course_enrollments.items():
            if student_id in enrolled_students:
                course = self.courses[course_id]
                courses.append(course)

        return courses
```

Система оценок и академическая успеваемость

Оценки представляют собой сложную вложенную структуру, демонстрирующую многоуровневую иерархию данных:

```
def add_grade(self, student_id: str, course_id: str,
                  assignment: str, score: float, max_score: float = 100.0):
        """Добавить оценку студенту"""
        # Проверяем, что студент зарегистрирован на курс
        if (course_id not in self.course_enrollments or
            student_id not in self.course_enrollments[course_id]):
            print(f"Student {student_id} is not enrolled in course {course_
id}")
            return False

        # Создаем запись об оценке
        grade_entry = {
            'assignment': assignment,
            'score': score,
            'max_score': max_score,
            'percentage': (score / max_score) * 100,
            'date': datetime.date.today().isoformat()
        }

        # Добавляем оценку в структуру
        if student_id not in self.grades:
            self.grades[student_id] = {}
        if course_id not in self.grades[student_id]:
            self.grades[student_id][course_id] = []

        self.grades[student_id][course_id].append(grade_entry)

        # Обновляем GPA студента
        self._update_student_gpa(student_id)

        print(f"Grade added for student {student_id} in course {course_id}")
        return True
    def get_student_grades(self, student_id: str, course_id: str = None) ->
Dict:
        """Получить оценки студента"""
        if student_id not in self.grades:
            return {}

        if course_id:
            # Оценки по конкретному курсу
            return self.grades[student_id].get(course_id, [])
        else:
            # Все оценки студента
            return self.grades[student_id]
    def calculate_course_average(self, student_id: str, course_id: str) ->
float:
```

```
        """Вычислить средний балл студента по курсу"""
        grades = self.get_student_grades(student_id, course_id)

        if not grades:
            return 0.0

        total_percentage = sum(grade['percentage'] for grade in grades)
        return total_percentage / len(grades)
    def _update_student_gpa(self, student_id: str):
        """Обновить общий GPA студента"""
        if student_id not in self.grades:
            return

        total_points = 0.0
        total_courses = 0

        for course_id, course_grades in self.grades[student_id].items():
            if course_grades:  # Есть оценки по курсу
               course_average = sum(g['percentage'] for g in course_grades)
/ len(course_grades)
                total_points += course_average
                total_courses += 1

        if total_courses > 0:
            gpa = total_points / total_courses / 25.0  # Конвертация в 4.0
систему
            student = self.find_student(student_id)
            if student:
                student['gpa'] = round(gpa, 2)
```

Расписание и управление временем

Расписание демонстрирует использование кортежей для неизменяемых временных данных:

```
def add_schedule_entry(self, course_id: str, day_of_week: str,
                       start_time: str, end_time: str, room: str):
        """Добавить занятие в расписание"""
        if course_id not in self.courses:
            print(f"Course {course_id} not found")
            return False

        # Создаем неизменяемую запись расписания (кортеж)
        schedule_entry = (
            course_id,
            day_of_week.lower(),
            start_time,
            end_time,
            room,
            datetime.date.today().isoformat()  # Дата добавления
        )
```

```
        self.schedule.append(schedule_entry)
        print(f"Schedule entry added for course {course_id}")
        return True
    def get_schedule_by_day(self, day_of_week: str) -> List[Dict]:
        """Получить расписание на определенный день"""
        day_lower = day_of_week.lower()
        day_schedule = []

        for entry in self.schedule:
            course_id, day, start_time, end_time, room, _ = entry

            if day == day_lower:
                course = self.courses[course_id]
                day_schedule.append({
                    'course_name': course['name'],
                    'course_id': course_id,
                    'instructor': course['instructor'],
                    'time': f"{start_time} - {end_time}",
                    'room': room
                })

        # Сортируем по времени начала
        day_schedule.sort(key=lambda x: x['time'])
        return day_schedule
    def get_student_schedule(self, student_id: str) -> Dict[str, List]:
        """Получить персональное расписание студента"""
        student_courses = self.get_student_courses(student_id)
        student_course_ids = {course['id'] for course in student_courses}

        weekly_schedule = {
            'monday': [], 'tuesday': [], 'wednesday': [],
            'thursday': [], 'friday': [], 'saturday': [], 'sunday': []
        }

        for entry in self.schedule:
            course_id, day, start_time, end_time, room, _ = entry

            if course_id in student_course_ids:
                course = self.courses[course_id]
                class_info = {
                    'course_name': course['name'],
                    'instructor': course['instructor'],
                    'time': f"{start_time} - {end_time}",
                    'room': room
                }
                weekly_schedule[day].append(class_info)

        return weekly_schedule
```

Аналитика и отчеты

Завершим систему мощными аналитическими функциями, демонстрирующими комплексную обработку данных:

```
def generate_student_report(self, student_id: str) -> Dict:
        """Сгенерировать полный отчет по студенту"""
        student = self.find_student(student_id)
        if not student:
            return {}

        # Базовая информация
        report = {
            'student_info': student.copy(),
            'enrolled_courses': [],
            'academic_performance': {},
            'schedule': self.get_student_schedule(student_id)
        }
                # Информация о курсах и успеваемости
        student_courses = self.get_student_courses(student_id)
        for course in student_courses:
            course_id = course['id']
                course_average = self.calculate_course_average(student_id,
course_id)

            course_info = {
                'course_name': course['name'],
                'instructor': course['instructor'],
                'credits': course['credits'],
                'current_average': round(course_average, 2),
              'assignments_completed': len(self.get_student_grades(student_
id, course_id))
            }

            report['enrolled_courses'].append(course_info)

        # Общая академическая статистика
        report['academic_performance'] = {
            'total_courses': len(student_courses),
            'current_gpa': student['gpa'],
             'total_credits': sum(course['credits'] for course in student_
courses)
        }

        return report
    def generate_course_statistics(self, course_id: str) -> Dict:
        """Сгенерировать статистику по курсу"""
        if course_id not in self.courses:
            return {}

        course = self.courses[course_id]
```

```
        enrolled_students = self.get_course_students(course_id)

        statistics = {
            'course_info': course,
            'enrollment': {
                'total_students': len(enrolled_students),
                 'capacity_used': f"{len(enrolled_students)}/{course['max_
students']}"
            },
            'grade_statistics': {}
        }

        # Статистика оценок
        all_averages = []
        for student in enrolled_students:
            student_id = student['id']
            average = self.calculate_course_average(student_id, course_id)
            if average > 0:
                all_averages.append(average)

        if all_averages:
            statistics['grade_statistics'] = {
              'class_average': round(sum(all_averages) / len(all_averages),
2),
                'highest_grade': round(max(all_averages), 2),
                'lowest_grade': round(min(all_averages), 2),
                'students_with_grades': len(all_averages)
            }

        return statistics
    def find_top_students(self, limit: int = 5) -> List[Dict]:
        """Найти лучших студентов по GPA"""
          active_students = [s for s in self.students if s['active'] and
s['gpa'] > 0]

        # Сортируем по GPA в убывающем порядке
           top_students = sorted(active_students, key=lambda x: x['gpa'],
reverse=True)

        return top_students[:limit]
    def get_system_overview(self) -> Dict:
        """Получить общий обзор системы"""
        active_students = [s for s in self.students if s['active']]
        active_courses = [c for c in self.courses.values() if c['active']]

        # Подсчет студентов по специальностям
        major_counts = {}
        for student in active_students:
            major = student['major']
            major_counts[major] = major_counts.get(major, 0) + 1
```

```
        return {
            'total_students': len(active_students),
            'total_courses': len(active_courses),
            'students_by_major': major_counts,
            'average_gpa': round(
                sum(s['gpa'] for s in active_students if s['gpa'] > 0) /
                len([s for s in active_students if s['gpa'] > 0]), 2
            ) if any(s['gpa'] > 0 for s in active_students) else 0,
              'total_enrollments': sum(len(enrolled) for enrolled in self.
course_enrollments.values())
        }
```

Демонстрация системы в действии

Теперь продемонстрируем полную функциональность системы:

```
def demonstrate_student_system():
    """Демонстрация работы системы управления студентами"""

    # Создаем экземпляр системы
    sms = StudentManagementSystem()

    print("=== Student Management System Demo ===\n")

    # Добавляем студентов
    print("1. Adding students:")
     alice_id = sms.add_student("Alice Johnson", "alice@email.com", "2000-
05-15", "Computer Science")
     bob_id = sms.add_student("Bob Smith", "bob@email.com", "1999-08-22",
"Mathematics")
      charlie_id = sms.add_student("Charlie Brown", "charlie@email.com",
"2001-01-10", "Computer Science")

    # Добавляем курсы
    print("\n2. Adding courses:")
    python_course = sms.add_course("Python Programming", "Dr. Sarah Wilson", 3)
    math_course = sms.add_course("Calculus I", "Prof. John Miller", 4)
    algo_course = sms.add_course("Algorithms", "Dr. Emily Davis", 3)

    # Регистрируем студентов на курсы
    print("\n3. Enrolling students:")
    sms.enroll_student(alice_id, python_course)
    sms.enroll_student(alice_id, math_course)
    sms.enroll_student(alice_id, algo_course)
    sms.enroll_student(bob_id, python_course)
    sms.enroll_student(bob_id, math_course)
    sms.enroll_student(charlie_id, python_course)
    sms.enroll_student(charlie_id, algo_course)

    # Добавляем расписание
    print("\n4. Adding schedule:")
```

```
    sms.add_schedule_entry(python_course, "Monday", "09:00", "10:30", "Room
101")
      sms.add_schedule_entry(python_course, "Wednesday", "09:00", "10:30",
"Room 101")
     sms.add_schedule_entry(math_course, "Tuesday", "14:00", "15:30", "Room
205")
    sms.add_schedule_entry(math_course, "Thursday", "14:00", "15:30", "Room
205")
     sms.add_schedule_entry(algo_course, "Friday", "11:00", "12:30", "Room
303")

    # Добавляем оценки
    print("\n5. Adding grades:")
    sms.add_grade(alice_id, python_course, "Homework 1", 95, 100)
    sms.add_grade(alice_id, python_course, "Quiz 1", 88, 100)
    sms.add_grade(alice_id, math_course, "Midterm", 92, 100)
    sms.add_grade(bob_id, python_course, "Homework 1", 87, 100)
    sms.add_grade(bob_id, math_course, "Midterm", 84, 100)
    sms.add_grade(charlie_id, python_course, "Homework 1", 91, 100)
    sms.add_grade(charlie_id, algo_course, "Project 1", 96, 100)

    # Генерируем отчеты
    print("\n6. Generating reports:")

    # Отчет по студенту
    alice_report = sms.generate_student_report(alice_id)
    print(f"\nAlice's Report:")
    print(f"GPA: {alice_report['student_info']['gpa']}")
      print(f"Total courses: {alice_report['academic_performance']['total_
courses']}")
    print("Enrolled courses:")
    for course in alice_report['enrolled_courses']:
        print(f"  - {course['course_name']}: {course['current_average']}%")

    # Статистика по курсу
    python_stats = sms.generate_course_statistics(python_course)
    print(f"\nPython Course Statistics:")
          print(f"Enrolled  students:  {python_stats['enrollment']['total_
students']}")
    if python_stats['grade_statistics']:
          print(f"Class average: {python_stats['grade_statistics']['class_
average']}%")

    # Расписание студента
    alice_schedule = sms.get_student_schedule(alice_id)
    print(f"\nAlice's Monday Schedule:")
    for class_info in alice_schedule['monday']:
          print(f"   {class_info['time']}: {class_info['course_name']} in
{class_info['room']}")

    # Общий обзор системы
```

```
    overview = sms.get_system_overview()
    print(f"\nSystem Overview:")
    print(f"Total students: {overview['total_students']}")
    print(f"Total courses: {overview['total_courses']}")
    print(f"Average GPA: {overview['average_gpa']}")
    print(f"Students by major: {overview['students_by_major']}")

    # Топ-студенты
    top_students = sms.find_top_students(3)
    print(f"\nTop Students:")
    for i, student in enumerate(top_students, 1):
            print(f"   {i}. {student['name']} ({student['major']}) - GPA:
{student['gpa']}")
# Запускаем демонстрацию
if __name__ == "__main__":
    demonstrate_student_system()
```

Анализ архитектурных решений

Этот проект демонстрирует несколько ключевых принципов работы со структурами данных.

Выбор структур для оптимизации операций: мы используем словарь `student_index` для быстрого поиска студентов по ID (O(1)) вместо линейного поиска по списку (O(n)). Множества в `course_enrollments` автоматически обеспечивают уникальность регистраций и быструю проверку принадлежности.

Иерархическая организация данных: структура оценок `grades[student_id][course_id][]` естественно отражает реальные отношения между студентами, курсами и оценками. Это делает код интуитивно понятным и эффективным.

Неизменяемость для стабильных данных: записи расписания представлены кортежами, поскольку после создания они редко изменяются. Это защищает от случайных модификаций и подчеркивает неизменный характер данных.

Денормализация для производительности: мы дублируем GPA в записи студента, хотя его можно вычислить из оценок. Это классический компромисс между консистентностью данных и скоростью доступа.

Расширения и улучшения

Созданная система представляет собой прочный фундамент, который можно расширить в различных направлениях:

```
# Возможные расширения системы:
# 1. Добавление системы предпосылок для курсов
def add_course_prerequisites(self, course_id: str, prerequisite_courses:
List[str]):
    """Добавить предпосылки для курса"""
    pass
# 2. Система уведомлений и дедлайнов
def add_assignment_deadline(self, course_id: str, assignment: str, deadline:
str):
    """Добавить дедлайн для задания"""
    pass
```

```
# 3. Экспорт данных в различные форматы
def export_to_json(self, filename: str):
    """Экспортировать данные в JSON-файл"""
    pass
# 4. Система резервного копирования
def create_backup(self):
    """Создать резервную копию системы"""
    Pass
```

Система учета студентов демонстрирует, как правильно спроектированные структуры данных становятся основой для создания масштабируемых и поддерживаемых приложений. Каждая структура выбрана осознанно, исходя из паттернов доступа к данным и требований производительности.

Этот проект объединяет все изученные концепции: от простых списков и словарей до сложных вложенных структур и аналитических алгоритмов. Понимание принципов, заложенных в этой системе, позволит вам создавать эффективные решения для любых задач управления данными в Python.

Глава 7. Работа с файлами и данными

7.1. Файловая система глазами программиста

Любая программа, которую мы пишем, рано или поздно сталкивается с необходимостью сохранить данные или считать информацию извне. Файловая система — это способ организации данных на диске, и понимание ее принципов критически важно для любого программиста. В этом разделе мы изучим, как Python взаимодействует с файловой системой, и научимся эффективно управлять файлами и папками.

Концепция файловой системы

Файловая система представляет собой иерархическую структуру, где данные организованы в виде файлов, расположенных в директориях (папках). Для программиста важно понимать, что файл — это именованная последовательность байтов, хранящаяся на диске. Python абстрагирует множество низкоуровневых деталей работы с файловой системой, предоставляя удобные инструменты для работы с файлами.

В операционных системах существуют различные типы файлов: текстовые файлы содержат читаемый текст, двоичные файлы хранят данные в специальных форматах, исполняемые файлы содержат программы. Для начинающего программиста наиболее важны текстовые файлы, поскольку они позволяют легко просматривать и редактировать содержимое.

Пути к файлам: абсолютные и относительные

Путь к файлу (path) — это адрес, по которому операционная система может найти конкретный файл. Существуют два типа путей: абсолютные и относительные.

Абсолютный путь указывает полное местоположение файла от корня файловой системы. В Windows он начинается с буквы диска, например: `C:\Users\username\Documents\data.txt`. В Unix-подобных системах (Linux, macOS) абсолютный путь начинается с косой черты: `/home/username/documents/data.txt`.

Относительный путь указывает местоположение файла относительно текущей рабочей директории (working directory). Если ваша программа находится в папке `projects`, а файл `data.txt` — в той же папке, относительный путь будет просто `data.txt`.

Python предоставляет модуль `os` для работы с операционной системой и модуль `pathlib` для современной работы с путями:

```
import os
from pathlib import Path
# Получаем текущую рабочую директорию
current_dir = os.getcwd()
print(f"Текущая директория: {current_dir}")
# Создаем путь с помощью pathlib (рекомендуемый способ)
file_path = Path("data") / "input.txt"
print(f"Путь к файлу: {file_path}")
```

```
# Проверяем существование файла
if file_path.exists():
    print("Файл существует")
else:
    print("Файл не найден")
```

Работа с путями в Python

Модуль pathlib, появившийся в Python 3.4, предоставляет объектно-ориентированный интерфейс для работы с файловыми путями. Это современный и предпочтительный способ работы с файлами:

```
from pathlib import Path
# Создание объекта пути
project_dir = Path("my_project")
data_file = project_dir / "data" / "results.txt"
# Получение различных частей пути
print(f"Имя файла: {data_file.name}")
print(f"Расширение: {data_file.suffix}")
print(f"Имя без расширения: {data_file.stem}")
print(f"Родительская директория: {data_file.parent}")
# Создание абсолютного пути
abs_path = data_file.resolve()
print(f"Абсолютный путь: {abs_path}")
```

Операции с файловой системой

Python позволяет выполнять основные операции с файловой системой — создание директорий, проверка существования файлов, получение информации о файлах:

```
from pathlib import Path
import os
# Создание директории
data_dir = Path("project_data")
data_dir.mkdir(exist_ok=True)  # exist_ok=True предотвращает ошибку, если
папка уже существует
# Создание вложенных директорий
nested_dir = Path("project_data/analysis/results")
nested_dir.mkdir(parents=True, exist_ok=True)
# Получение списка файлов в директории
for file_path in data_dir.iterdir():
    if file_path.is_file():
        print(f"Файл: {file_path.name}")
    elif file_path.is_dir():
        print(f"Директория: {file_path.name}")
# Получение информации о файле
if data_dir.exists():
    stat_info = data_dir.stat()
    print(f"Размер: {stat_info.st_size} байт")
    print(f"Время модификации: {stat_info.st_mtime}")
```

Понимание файловой системы критически важно для написания надежных программ. Всегда проверяйте существование файлов перед попыткой их открытия, используйте относительные пути для переносимости программы между системами и помните о различиях в разделителях путей между операционными системами.

7.2. Чтение и запись текстовых файлов

Работа с текстовыми файлами — одна из самых распространенных задач в программировании. Текстовые файлы используются для хранения конфигураций, данных, логов, результатов вычислений. В этом разделе мы изучим, как Python открывает, читает и записывает текстовые файлы.

Основы работы с файлами: функция open()

Центральная функция для работы с файлами в Python — это `open()`. Она возвращает объект файла, с помощью которого можно выполнять операции чтения и записи:

```
# Простейший способ открытия файла для чтения
file = open('example.txt', 'r', encoding='utf-8')
content = file.read()
print(content)
file.close()  # Важно закрыть файл после использования
```

Функция `open()` принимает несколько важных параметров:

- **`filename`** — имя файла (строка или объект `Path`);
- **`mode`** — режим открытия файла (строка);
- **`encoding`** — кодировка для текстовых файлов (по умолчанию зависит от системы).

Режимы открытия файлов

Режим открытия определяет, что вы можете делать с файлом. Основные режимы:

- `'r'` — чтение (`read`). Файл должен существовать;
- `'w'` — запись (`write`). Создает новый файл или перезаписывает существующий;
- `'a'` — добавление (`append`). Добавляет данные в конец существующего файла;
- `'x'` — эксклюзивное создание. Создает файл, только если он не существует;
- `'r+'` — чтение и запись.

```
# Различные режимы открытия файлов
from pathlib import Path
# Запись в файл (перезапись содержимого)
with open('output.txt', 'w', encoding='utf-8') as file:
    file.write('Первая строка\n')
    file.write('Вторая строка\n')
# Добавление в существующий файл
with open('output.txt', 'a', encoding='utf-8') as file:
    file.write('Третья строка\n')
# Чтение файла
with open('output.txt', 'r', encoding='utf-8') as file:
```

```
    content = file.read()
    print(content)
```

Конструкция `with`: безопасная работа с файлами

Использование конструкции `with` (контекстный менеджер) является лучшей практикой при работе с файлами. Она автоматически закрывает файл даже в случае возникновения ошибки:

```
# Рекомендуемый способ работы с файлами
with open('data.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
# Файл автоматически закрывается здесь
```

Способы чтения файлов

Python предоставляет несколько способов чтения файлов, каждый из которых подходит для определенных сценариев:

```
# Создаем тестовый файл
test_data = """Первая строка файла
Вторая строка
Третья строка
Четвертая строка"""
with open('test.txt', 'w', encoding='utf-8') as file:
    file.write(test_data)
# Способ 1: чтение всего файла целиком
print("=== Чтение всего файла ===")
with open('test.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(repr(content))  # repr показывает специальные символы
# Способ 2: чтение файла построчно в список
print("\n=== Чтение в список строк ===")
with open('test.txt', 'r', encoding='utf-8') as file:
    lines = file.readlines()
    for i, line in enumerate(lines):
        print(f"Строка {i}: {repr(line)}")
# Способ 3: чтение файла построчно в цикле (наиболее эффективный)
print("\n=== Чтение построчно в цикле ===")
with open('test.txt', 'r', encoding='utf-8') as file:
    for line_num, line in enumerate(file, 1):
        print(f"Строка {line_num}: {line.strip()}")
```

Запись в файлы

Для записи данных в файлы используются методы `write()` и `writelines()`:

```
# Запись строк по одной
data_to_write = ['Первая строка данных', 'Вторая строка данных', 'Третья
строка данных']
with open('output_data.txt', 'w', encoding='utf-8') as file:
```

```
    for line in data_to_write:
        file.write(line + '\n')  # Добавляем перевод строки вручную
# Запись списка строк сразу
with open('output_data2.txt', 'w', encoding='utf-8') as file:
    # Добавляем '\n' к каждой строке
    lines_with_newline = [line + '\n' for line in data_to_write]
    file.writelines(lines_with_newline)
# Запись форматированных данных
students = [
    {'name': 'Иван', 'grade': 85},
    {'name': 'Мария', 'grade': 92},
    {'name': 'Петр', 'grade': 78}
]
with open('students.txt', 'w', encoding='utf-8') as file:
    file.write('Список студентов и их оценки:\n')
    file.write('-' * 30 + '\n')
    for student in students:
        file.write(f"{student['name']}: {student['grade']}\n")
```

Обработка кодировок

Кодировка определяет, как символы представляются в виде байтов. Для русскоязычных текстов почти всегда используется UTF-8:

```
# Правильная работа с русскими символами
russian_text = "Привет, мир! Это текст на русском языке."
# Запись с явным указанием кодировки
with open('russian.txt', 'w', encoding='utf-8') as file:
    file.write(russian_text)
# Чтение с правильной кодировкой
with open('russian.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

Практический пример: анализатор текста

Давайте создадим практическую программу, которая анализирует текстовый файл и выдает статистику:

```
def analyze_text_file(filename):
    """
    Анализирует текстовый файл и возвращает статистику
    """
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            lines = file.readlines()

        # Подсчет базовой статистики
        total_lines = len(lines)
        total_chars = sum(len(line) for line in lines)
        total_words = sum(len(line.split()) for line in lines)
        non_empty_lines = sum(1 for line in lines if line.strip())
```

```
                # Поиск самой длинной строки
            longest_line = max(lines, key=len) if lines else ""

            return {
                'lines': total_lines,
                'non_empty_lines': non_empty_lines,
                'characters': total_chars,
                'words': total_words,
                'longest_line': longest_line.strip()
            }

        except FileNotFoundError:
            print(f"Файл {filename} не найден")
            return None
        except Exception as e:
            print(f"Ошибка при чтении файла: {e}")
            return None
# Создаем тестовый файл
sample_text = """Python — это высокоуровневый язык программирования.
Он отличается простым и понятным синтаксисом.
Python широко используется в различных областях:
- Веб-разработка
- Анализ данных
- Машинное обучение
- Автоматизация задач"""
with open('sample.txt', 'w', encoding='utf-8') as file:
    file.write(sample_text)
# Анализируем файл
stats = analyze_text_file('sample.txt')
if stats:
    print("=== Статистика текстового файла ===")
    print(f"Всего строк: {stats['lines']}")
    print(f"Непустых строк: {stats['non_empty_lines']}")
    print(f"Символов: {stats['characters']}")
    print(f"Слов: {stats['words']}")
    print(f"Самая длинная строка: '{stats['longest_line']}'")
```

Работа с текстовыми файлами — фундаментальный навык программиста. Всегда используйте конструкцию `with` для автоматического закрытия файлов, явно указывайте кодировку UTF-8 для работы с русским текстом и помните о необходимости обработки ошибок при работе с файлами. В следующих разделах мы изучим более специализированные форматы данных, такие как CSV и JSON.

7.3. Обработка CSV-данных

Формат CSV (Comma-Separated Values — значения, разделенные запятыми) является одним из наиболее распространенных способов хранения табличных данных. CSV-файлы используются повсеместно: от экспорта данных из электронных таблиц до обмена информацией между различными системами. Понимание работы с CSV критически важно для любого программиста, занимающегося обработкой данных.

Что представляет собой формат CSV

CSV — это текстовый формат, в котором каждая строка представляет запись (строку таблицы), а значения в записи разделены специальными символами, чаще всего запятыми. Простота формата делает его универсальным: CSV-файлы можно открывать в любом текстовом редакторе, электронных таблицах и практически всех системах управления базами данных.

Рассмотрим простой пример CSV-файла с информацией о студентах:

```
name,age,grade,city
Иван Петров,20,85,Москва
Мария Сидорова,19,92,Санкт-Петербург
Алексей Иванов,21,78,Казань
Елена Козлова,20,96,Новосибирск
```

Первая строка содержит заголовки столбцов (header), а последующие строки — данные. Каждое поле отделено запятой, что позволяет легко определить структуру данных.

Модуль csv: стандартные инструменты Python

Python предоставляет встроенный модуль `csv`, который значительно упрощает работу с CSV-файлами. Этот модуль правильно обрабатывает различные особенности формата: кавычки, запятые внутри значений, различные разделители:

```
import csv
from pathlib import Path
# Создадим тестовый CSV-файл
sample_data = '''name,age,grade,city
Иван Петров,20,85,Москва
Мария Сидорова,19,92,Санкт-Петербург
Алексей Иванов,21,78,Казань
Елена Козлова,20,96,Новосибирск'''
with open('students.csv', 'w', encoding='utf-8') as file:
    file.write(sample_data)
print("CSV-файл создан успешно")
```

Чтение CSV-файлов

Модуль `csv` предоставляет несколько способов чтения данных. Наиболее распространенные — это `csv.reader()` для простого чтения и `csv.DictReader()` для чтения с именованными полями:

```
# Простое чтение CSV с помощью csv.reader
print("=== Чтение с помощью csv.reader ===")
with open('students.csv', 'r', encoding='utf-8') as file:
    csv_reader = csv.reader(file)
    headers = next(csv_reader)  # Читаем заголовки отдельно
    print(f"Заголовки: {headers}")

    for row_num, row in enumerate(csv_reader, 1):
        print(f"Строка {row_num}: {row}")
# Чтение CSV с помощью csv.DictReader (более удобно)
```

```
print("\n=== Чтение с помощью csv.DictReader ===")
with open('students.csv', 'r', encoding='utf-8') as file:
    csv_reader = csv.DictReader(file)
    print(f"Имена полей: {csv_reader.fieldnames}")

    for row in csv_reader:
            print(f"Студент: {row['name']}, Возраст: {row['age']}, Оценка:
{row['grade']}")
```

Запись CSV-файлов

Для записи CSV-данных используются `csv.writer()` и `csv.DictWriter()`. Второй вариант особенно удобен при работе со структурированными данными:

```
# Подготовка данных для записи
students_data = [
   {'name': 'Анна Волкова', 'age': 22, 'grade': 88, 'city': 'Екатеринбург'},
   {'name': 'Дмитрий Смирнов', 'age': 20, 'grade': 91, 'city': 'Челябинск'},
   {'name': 'Ольга Петрова', 'age': 19, 'grade': 87, 'city': 'Ростов-на-До-
ну'}
]
# Запись с помощью csv.DictWriter
with open('new_students.csv', 'w', encoding='utf-8', newline='') as file:
    fieldnames = ['name', 'age', 'grade', 'city']
    csv_writer = csv.DictWriter(file, fieldnames=fieldnames)

    csv_writer.writeheader()  # Записываем заголовки
    for student in students_data:
        csv_writer.writerow(student)
print("Новый CSV-файл записан")
# Запись с помощью csv.writer (более низкоуровневый подход)
with open('simple_data.csv', 'w', encoding='utf-8', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['product', 'price', 'quantity'])  # Заголовки
    csv_writer.writerow(['Ноутбук', 45000, 5])
    csv_writer.writerow(['Мышь', 800, 20])
    csv_writer.writerow(['Клавиатура', 2500, 15])
```

Обработка различных форматов CSV

Реальные CSV-файлы часто отклоняются от стандарта: используют другие разделители, различные способы экранирования кавычек. Модуль `csv` позволяет настроить обработку под конкретный формат:

```
# Создаем файл с точкой с запятой в качестве разделителя
european_csv = '''name;age;salary;department
"Иванов, Пётр";30;"50,000";IT
"Сидорова; Мария";25;"45,500";HR
"Петров, Алексей";35;"60,000";Finance'''
with open('european.csv', 'w', encoding='utf-8') as file:
    file.write(european_csv)
```

```
# Чтение файла с нестандартным разделителем
print("=== Чтение файла с разделителем ; ===")
with open('european.csv', 'r', encoding='utf-8') as file:
    csv_reader = csv.DictReader(file, delimiter=';')
    for row in csv_reader:
        print(f"Сотрудник: {row['name']}, Отдел: {row['department']}, Зар-
плата: {row['salary']}")
# Настройка диалекта CSV для повторного использования
csv.register_dialect('semicolon',       delimiter=';',       quotechar='"',
skipinitialspace=True)
with open('european.csv', 'r', encoding='utf-8') as file:
    csv_reader = csv.DictReader(file, dialect='semicolon')
    for row in csv_reader:
        print(f"Имя: {row['name']}, Возраст: {row['age']}")
```

7.4. Работа с JSON: современный формат данных

JSON (JavaScript Object Notation — объектная нотация JavaScript) фактически стал стандартом для обмена данными в современных приложениях. Несмотря на происхождение от JavaScript, JSON является языково-независимым форматом, который поддерживается практически всеми современными языками программирования, включая Python.

Что такое JSON и почему он важен

JSON представляет данные в виде пар «ключ-значение», используя синтаксис, очень похожий на словари Python. Это делает его естественным выбором для Python-разработчиков. JSON широко используется в веб-API, конфигурационных файлах, обмене данными между сервисами и хранении структурированной информации.

Основные преимущества JSON:

- человекочитаемый формат;
- лёгкий парсинг и генерация;
- поддержка вложенных структур;
- компактность по сравнению с XML;
- нативная поддержка в JavaScript.

Пример JSON-документа:

```
{
  "name": "Иван Петров",
  "age": 25,
  "is_student": true,
  "courses": ["Python", "JavaScript", "SQL"],
  "address": {
    "city": "Москва",
    "street": "Тверская",
    "building": 10
  },
  "phone": null
}
```

Модуль `json`: встроенные возможности Python

Python предоставляет встроенный модуль `json`, который позволяет легко преобразовывать Python-объекты в JSON-строки (сериализация) и обратно (десериализация):

```
import json
# Python-объект для демонстрации
student_data = {
    'name': 'Мария Сидорова',
    'age': 22,
    'is_graduate': False,
    'grades': [85, 90, 78, 92],
    'contact': {
        'email': 'maria@example.com',
        'phone': '+7-123-456-7890'
    },
    'graduation_year': None
}
# Сериализация: Python объект -> JSON строка
json_string = json.dumps(student_data, ensure_ascii=False, indent=2)
print("=== Сериализованный JSON ===")
print(json_string)
print(f"Тип: {type(json_string)}")
# Десериализация: JSON-строка -> Python-объект
parsed_data = json.loads(json_string)
print("\n=== Десериализованный объект ===")
print(f"Имя: {parsed_data['name']}")
print(f"Оценки: {parsed_data['grades']}")
print(f"Тип: {type(parsed_data)}")
```

Соответствие типов данных

При работе с JSON важно понимать, как типы данных Python соответствуют типам JSON:

Python → JSON:

- `dict` → объект `{}`
- `list`, `tuple` → массив `[]`
- `str` → строка
- `int`, `float` → число
- `True`, `False` → `true`, `false`
- `None` → `null`

JSON → Python:

- объект `{}` → `dict`
- массив `[]` → `list`
- строка → `str`
- число → `int` или `float`
- `true`, `false` → `True`, `False`
- `null` → `None`

```
# Демонстрация преобразования типов
test_data = {
    'string': 'Привет, мир!',
    'integer': 42,
    'float': 3.14159,
    'boolean_true': True,
    'boolean_false': False,
    'null_value': None,
    'list': [1, 2, 3, 'четыре'],
    'nested_dict': {
        'inner_key': 'inner_value',
        'inner_list': [True, False, None]
    }
}
# Сериализация и десериализация
json_repr = json.dumps(test_data, ensure_ascii=False, indent=2)
restored_data = json.loads(json_repr)
print("Исходные типы:")
for key, value in test_data.items():
    print(f"  {key}: {type(value).__name__}")
print("\nВосстановленные типы:")
for key, value in restored_data.items():
    print(f"  {key}: {type(value).__name__}")
```

Работа с JSON-файлами

Для работы с JSON-файлами модуль предоставляет функции `dump()` и `load()`, которые работают непосредственно с файлами:

```
# Подготовка данных о команде разработчиков
development_team = {
    'project_name': 'Система управления задачами',
    'start_date': '2024-01-15',
    'team_lead': {
        'name': 'Анна Козлова',
        'experience_years': 8,
        'skills': ['Python', 'Django', 'PostgreSQL']
    },
    'developers': [
        {
            'name': 'Дмитрий Волков',
            'role': 'Backend Developer',
            'experience_years': 5,
            'skills': ['Python', 'FastAPI', 'Redis']
        },
        {
            'name': 'Елена Смирнова',
            'role': 'Frontend Developer',
            'experience_years': 3,
            'skills': ['JavaScript', 'React', 'CSS']
        }
    ],
```

```
    'technologies': ['Python', 'JavaScript', 'PostgreSQL', 'Redis', 'Docker'],
     'is_active': True
}
# Запись в JSON-файл
with open('team_data.json', 'w', encoding='utf-8') as file:
    json.dump(development_team, file, ensure_ascii=False, indent=2)
print("Данные команды сохранены в team_data.json")
# Чтение из JSON-файла
with open('team_data.json', 'r', encoding='utf-8') as file:
    loaded_team = json.load(file)
print(f"\nПроект: {loaded_team['project_name']}")
print(f"Руководитель: {loaded_team['team_lead']['name']}")
print(f"Количество разработчиков: {len(loaded_team['developers'])}")
print("\nСостав команды:")
for dev in loaded_team['developers']:
    print(f"  {dev['name']} - {dev['role']}")
```

Обработка ошибок при работе с JSON

При работе с JSON важно правильно обрабатывать возможные ошибки, особенно при парсинге данных из внешних источников:

```
def safe_json_parse(json_string):
    """Безопасный парсинг JSON с обработкой ошибок"""
    try:
        return json.loads(json_string), None
    except json.JSONDecodeError as e:
        return None, f"Ошибка парсинга JSON: {e}"
    except Exception as e:
        return None, f"Неожиданная ошибка: {e}"
# Тестирование с корректными и некорректными данными
test_cases = [
    '{"name": "Тест", "value": 123}',  # Корректный JSON
    '{"name": "Тест", "value": }',     # Некорректный JSON
    '{"name": "Тест"',                 # Незакрытые скобки
    'not json at all'                  # Вообще не JSON
]
for i, test_json in enumerate(test_cases, 1):
    print(f"\n=== Тест {i} ===")
    print(f"Входные данные: {test_json}")
    result, error = safe_json_parse(test_json)

    if result is not None:
        print(f"Успешно распознано: {result}")
    else:
        print(f"Ошибка: {error}")
def safe_json_file_load(filename):
    """Безопасная загрузка JSON из файла"""
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            return json.load(file), None
```

```
    except FileNotFoundError:
        return None, f"Файл {filename} не найден"
    except json.JSONDecodeError as e:
        return None, f"Файл содержит некорректный JSON: {e}"
    except Exception as e:
        return None, f"Ошибка при чтении файла: {e}"
```

7.5. Обработка исключений: что делать, когда что-то идет не так

В реальном программировании ошибки неизбежны. Файлы могут отсутствовать, сетевые соединения — прерываться, пользователи — вводить некорректные данные. **Обработка исключений (exceptions)** — это механизм, позволяющий программе реагировать на ошибочные ситуации, не завершая работу аварийно. Понимание системы исключений критически важно для написания надежного кода.

Философия обработки ошибок в Python

Python следует принципу «Легче просить прощения, чем разрешения» (EAFP — Easier to Ask for Forgiveness than Permission). Это означает, что вместо предварительной проверки всех возможных условий лучше попытаться выполнить операцию и обработать возможную ошибку.

Рассмотрим разность подходов:

```
import os
# Подход "Сначала спроси разрешения" (LBYL - Look Before You Leap)
filename = 'data.txt'
if os.path.exists(filename):
    if os.access(filename, os.R_OK):
        with open(filename, 'r') as file:
            content = file.read()
            print(content)
    else:
        print("Файл недоступен для чтения")
else:
    print("Файл не существует")
# Подход "Легче просить прощения" (EAFP) - предпочтительный в Python
try:
    with open(filename, 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Файл не найден")
except PermissionError:
    print("Недостаточно прав для чтения файла")
except Exception as e:
    print(f"Неожиданная ошибка: {e}")
```

Подход EAFP предпочтительнее, поскольку он более читаем, менее подвержен состоянию гонки (race conditions) и соответствует идеологии Python.

Базовая структура `try-except`

Основная конструкция для обработки исключений включает блоки `try`, `except`, `else` и `finally`:

```
def safe_division(a, b):
    """Демонстрирует полную структуру обработки исключений"""
    try:
        # Код, который может вызвать исключение
        result = a / b
        print(f"Вычисление {a} / {b} выполнено успешно")
    except ZeroDivisionError:
        # Обработка конкретного типа исключения
        print("Ошибка: деление на ноль!")
        return None
    except TypeError:
        # Обработка другого типа исключения
        print("Ошибка: некорректные типы данных для деления")
        return None
    except Exception as e:
        # Обработка всех остальных исключений
        print(f"Неожиданная ошибка: {e}")
        return None
    else:
        # Выполняется, только если не было исключений
        print("Деление завершено без ошибок")
        return result
    finally:
        # Выполняется всегда, независимо от наличия исключений
        print("Функция safe_division завершила работу")
# Тестирование различных сценариев
print("=== Тест 1: Нормальное деление ===")
result1 = safe_division(10, 2)
print(f"Результат: {result1}\n")
print("=== Тест 2: Деление на ноль ===")
result2 = safe_division(10, 0)
print(f"Результат: {result2}\n")
print("=== Тест 3: Некорректные типы ===")
result3 = safe_division("10", 2)
print(f"Результат: {result3}\n")
```

Иерархия исключений в Python

Python имеет богатую иерархию встроенных исключений. Понимание этой иерархии помогает писать более точную обработку ошибок:

```
# Демонстрация различных типов исключений при работе с файлами
import json
def demonstrate_file_exceptions():
    """Демонстрирует различные исключения при работе с файлами"""
```

```
    test_cases = [
        ('nonexistent_file.txt', 'r'),          # FileNotFoundError
        ('/root/protected_file.txt', 'w'),      # PermissionError (на Unix)
        ('existing_directory', 'r'),            # IsADirectoryError
    ]

    for filename, mode in test_cases:
        print(f"\n--- Попытка открыть {filename} в режиме '{mode}' ---")
        try:
            with open(filename, mode) as file:
                content = file.read()
                print("Файл успешно открыт")
        except FileNotFoundError:
            print("Файл не найден")
        except PermissionError:
            print("Недостаточно прав доступа")
        except IsADirectoryError:
            print("Указанный путь является директорией, а не файлом")
        except OSError as e:
            print(f"Ошибка операционной системы: {e}")
def demonstrate_json_exceptions():
    """Демонстрирует исключения при работе с JSON"""

    invalid_json_samples = [
        '{"name": "Тест"}',                     # Корректный JSON
        '{"name": "Тест",}',                    # Лишняя запятая
        '{"name": "Тест"',                      # Незакрытые скобки
        '{"name": Тест}',                       # Значение без кавычек
        '',                                     # Пустая строка
    ]

    for i, json_str in enumerate(invalid_json_samples, 1):
        print(f"\n--- JSON тест {i}: {json_str} ---")
        try:
            data = json.loads(json_str)
            print(f"JSON успешно разобран: {data}")
        except json.JSONDecodeError as e:
            print(f"Ошибка парсинга JSON: {e}")
            print(f"Позиция ошибки: строка {e.lineno}, колонка {e.colno}")
        except Exception as e:
            print(f"Неожиданная ошибка: {e}")
# Запуск демонстраций
demonstrate_file_exceptions()
demonstrate_json_exceptions()
```

Обработка множественных исключений

Python позволяет обрабатывать несколько типов исключений в одном блоке except или создавать цепочки обработчиков:

```
def process_user_input():
    """Демонстрирует обработку различных типов ошибок пользовательского ввода"""

    while True:
        try:
            # Имитируем ввод пользователя (в реальности используйте input())
            user_inputs = ["10", "abc", "0", "5.5", "quit"]

            for user_input in user_inputs:
                print(f"\nОбработка ввода: '{user_input}'")

                if user_input.lower() == 'quit':
                    print("Выход из программы")
                    return

                # Попытка преобразования в число
                number = int(user_input)

                # Операция, которая может вызвать деление на ноль
                result = 100 / number

                print(f"100 / {number} = {result}")

        except ValueError:
            print("Ошибка: введите корректное целое число")
        except ZeroDivisionError:
            print("Ошибка: деление на ноль невозможно")
        except (KeyboardInterrupt, EOFError):
            # Обработка нескольких типов исключений в одном блоке
            print("\nПрограмма прервана пользователем")
            break
        except Exception as e:
            print(f"Неожиданная ошибка: {type(e).__name__}: {e}")

        # В реальной программе здесь был бы continue для повтора цикла
process_user_input()
```

7.6. Практический проект: анализатор логов веб-сервера

Теперь мы готовы объединить все изученные навыки работы с файлами и данными в одном практическом проекте. Создадим анализатор логов веб-сервера — инструмент, который поможет веб-мастерам и системным администраторам извлекать полезную информацию из журналов работы веб-сервера.

Понимание задачи

Логи веб-сервера содержат записи о каждом запросе к сайту. Типичная строка лога Apache выглядит следующим образом:

```
192.168.1.100 - - [25/Dec/2024:10:15:30 +0000] "GET /index.html HTTP/1.1"
200 2326 "http://example.com/home" "Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36"
```

Наш анализатор должен извлекать из таких строк полезную статистику: самые популярные страницы, коды ответов сервера, активность по часам, географию пользователей по IP-адресам.

Архитектура решения

Разделим задачу на логические компоненты:

```
import re
import json
import csv
from collections import defaultdict, Counter
from datetime import datetime
import sys
class LogAnalyzer:
    """Анализатор логов веб-сервера"""

    def __init__(self, config_file="config.json"):
        """Инициализация с загрузкой конфигурации"""
        self.stats = {
            'total_requests': 0,
            'status_codes': Counter(),
            'popular_pages': Counter(),
            'hourly_traffic': defaultdict(int),
            'ip_addresses': Counter(),
            'user_agents': Counter()
        }
        self.load_config(config_file)

    def load_config(self, config_file):
        """Загрузка конфигурации из JSON-файла"""
        default_config = {
            'log_format': 'apache_common',
            'output_format': 'console',
            'top_results_count': 10,
            'exclude_static_files': True
        }

        try:
            with open(config_file, 'r', encoding='utf-8') as f:
                self.config = json.load(f)
        except FileNotFoundError:
             print(f"Файл конфигурации {config_file} не найден. Используем
настройки по умолчанию.")
```

```
            self.config = default_config
        except json.JSONDecodeError as e:
            print(f"Ошибка в файле конфигурации: {e}. Используем настройки
по умолчанию.")
            self.config = default_config
```

Парсинг строк лога

Создадим метод для разбора строк лога с использованием регулярных выражений:

```
def parse_log_line(self, line):
        """Парсинг строки лога Apache в формате Common Log Format"""
        # Регулярное выражение для Apache Common Log Format
        log_pattern = re.compile(
            r'(?P<ip>\d+\.\d+\.\d+\.\d+) '
            r'(?P<identity>\S+) '
            r'(?P<user>\S+) '
            r'\[(?P<datetime>[^\]]+)\] '
            r'"(?P<method>\S+) (?P<url>\S+) (?P<protocol>[^"]+)" '
            r'(?P<status>\d+) '
            r'(?P<size>\S+) '
            r'"(?P<referrer>[^"]*)" '
            r'"(?P<user_agent>[^"]*)"'
        )

        match = log_pattern.match(line.strip())
        if match:
            return match.groupdict()
        return None

    def should_include_request(self, parsed_line):
        """Проверка, следует ли включать запрос в анализ"""
        if not parsed_line:
            return False

        # Исключение статических файлов, если настроено
        if self.config.get('exclude_static_files', True):
              static_extensions = ['.css', '.js', '.png', '.jpg', '.gif',
'.ico', '.woff']
            url = parsed_line['url'].lower()
            if any(url.endswith(ext) for ext in static_extensions):
                return False

        return True
```

Анализ данных

Реализуем основную логику анализа:

```
def analyze_line(self, parsed_line):
```

```
        """Анализ одной строки лога"""
        if not self.should_include_request(parsed_line):
            return

        # Общая статистика
        self.stats['total_requests'] += 1

        # Коды ответов
        status_code = parsed_line['status']
        self.stats['status_codes'][status_code] += 1

        # Популярные страницы
        url = parsed_line['url']
        self.stats['popular_pages'][url] += 1

        # IP-адреса
        ip = parsed_line['ip']
        self.stats['ip_addresses'][ip] += 1

        # Пользовательские агенты
        user_agent = parsed_line['user_agent']
        if user_agent != '-':
            self.stats['user_agents'][user_agent] += 1

        # Трафик по часам
        try:
            # Парсинг даты: [25/Dec/2024:10:15:30 +0000]
            datetime_str = parsed_line['datetime']
        dt = datetime.strptime(datetime_str.split()[0], "%d/%b/%Y:%H:%M:%S")
            hour = dt.hour
            self.stats['hourly_traffic'][hour] += 1
        except ValueError:
            # Игнорируем строки с неправильным форматом даты
            pass

    def analyze_file(self, log_file):
        """Анализ файла логов"""
        line_count = 0
        error_count = 0

        try:
          with open(log_file, 'r', encoding='utf-8', errors='ignore') as f:
                for line_num, line in enumerate(f, 1):
                    line_count += 1

                    try:
                        parsed_line = self.parse_log_line(line)
                        if parsed_line:
                            self.analyze_line(parsed_line)
                        else:
                            error_count += 1
```

```
                    except Exception as e:
                        print(f"Ошибка в строке {line_num}: {e}")
                        error_count += 1

                    # Прогресс для больших файлов
                    if line_count % 10000 == 0:
                        print(f"Обработано строк: {line_count}")

        except FileNotFoundError:
            print(f"Файл {log_file} не найден.")
            return False
        except PermissionError:
            print(f"Нет доступа к файлу {log_file}.")
            return False

          print(f"Анализ завершен. Обработано строк: {line_count}, ошибок:
{error_count}")
        return True
```

Генерация отчетов

Создадим методы для вывода результатов в различных форматах:

```
def print_console_report(self):
        """Вывод отчета в консоль"""
        print("\n" + "="*50)
        print("ОТЧЕТ ПО АНАЛИЗУ ЛОГОВ ВЕБ-СЕРВЕРА")
        print("="*50)

        print(f"\nОбщая статистика:")
        print(f"Всего запросов: {self.stats['total_requests']:,}")

        # Коды ответов
        print(f"\nКоды ответов сервера:")
        for status_code, count in self.stats['status_codes'].most_common():
            percentage = (count / self.stats['total_requests']) * 100
            print(f"  {status_code}: {count:,} ({percentage:.1f}%)")

        # Популярные страницы
        top_count = self.config.get('top_results_count', 10)
        print(f"\nТоп-{top_count} популярных страниц:")
      for url, count in self.stats['popular_pages'].most_common(top_count):
            print(f"  {count:,} - {url}")

        # Активность по часам
        print(f"\nРаспределение трафика по часам:")
        for hour in range(24):
            count = self.stats['hourly_traffic'][hour]
            bar = "" * (count // max(1, max(self.stats['hourly_traffic'].
values()) // 50))
            print(f"  {hour:2d}:00 - {count:,} {bar}")
```

```
        # Топ IP-адресов
        print(f"\nТоп-{top_count} активных IP-адресов:")
        for ip, count in self.stats['ip_addresses'].most_common(top_count):
            print(f"  {count:,} - {ip}")

    def save_csv_report(self, filename):
        """Сохранение отчета в CSV-файлы"""
        try:
            # Популярные страницы
          with open(f"pages_{filename}", 'w', newline='', encoding='utf-8')
as f:
                writer = csv.writer(f)
                writer.writerow(['URL', 'Количество запросов', 'Процент'])

               for url, count in self.stats['popular_pages'].most_common():
                  percentage = (count / self.stats['total_requests']) * 100
                    writer.writerow([url, count, f"{percentage:.2f}%"])

            # IP-адреса
           with open(f"ips_{filename}", 'w', newline='', encoding='utf-8')
as f:
                writer = csv.writer(f)
                writer.writerow(['IP-адрес', 'Количество запросов'])

                for ip, count in self.stats['ip_addresses'].most_common():
                    writer.writerow([ip, count])

           print(f"CSV отчеты сохранены: pages_{filename}, ips_{filename}")

        except Exception as e:
            print(f"Ошибка при сохранении CSV: {e}")

    def save_json_report(self, filename):
        """Сохранение отчета в JSON-формате"""
        try:
            report_data = {
                'summary': {
                    'total_requests': self.stats['total_requests'],
                    'analysis_date': datetime.now().isoformat()
                },
                'status_codes': dict(self.stats['status_codes']),
                   'popular_pages': dict(self.stats['popular_pages'].most_
common(50)),
                'hourly_traffic': dict(self.stats['hourly_traffic']),
               'top_ips': dict(self.stats['ip_addresses'].most_common(20))
            }

            with open(filename, 'w', encoding='utf-8') as f:
                json.dump(report_data, f, ensure_ascii=False, indent=2)
```

```
            print(f"JSON отчет сохранен: {filename}")

        except Exception as e:
            print(f"Ошибка при сохранении JSON: {e}")
```

Главная функция и использование

Завершим проект главной функцией и примером использования:

```
def main():
    """Главная функция программы"""
    if len(sys.argv) < 2:
        print("Использование: python log_analyzer.py <файл_логов> [config.
json]")
        print("Пример: python log_analyzer.py access.log")
        return

    log_file = sys.argv[1]
    config_file = sys.argv[2] if len(sys.argv) > 2 else "config.json"

    # Создание анализатора
    analyzer = LogAnalyzer(config_file)

    # Анализ файла
    print(f"Начинаем анализ файла: {log_file}")
    if analyzer.analyze_file(log_file):
        # Вывод результатов
        if analyzer.config.get('output_format', 'console') == 'console':
            analyzer.print_console_report()

        # Сохранение дополнительных отчетов
        base_filename = log_file.replace('.log', '').replace('.txt', '')
        analyzer.save_csv_report(f"{base_filename}_report.csv")
        analyzer.save_json_report(f"{base_filename}_report.json")
if __name__ == "__main__":
    main()
```

Файл конфигурации

Создадим пример файла конфигурации `config.json`:

```
{
    "log_format": "apache_common",
    "output_format": "console",
    "top_results_count": 15,
    "exclude_static_files": true,
     "static_extensions": [".css", ".js", ".png", ".jpg", ".gif", ".ico",
".woff", ".ttf"],
    "report_sections": {
        "status_codes": true,
        "popular_pages": true,
```

```
        "hourly_traffic": true,
        "ip_analysis": true
    }
}
```

Тестирование и расширение

Для тестирования создайте файл test_access.log с несколькими строками:

```
192.168.1.100 - - [25/Dec/2024:10:15:30 +0000] "GET /index.html HTTP/1.1"
200 2326
192.168.1.101 - - [25/Dec/2024:10:16:45 +0000] "GET /about.html HTTP/1.1"
200 1825
192.168.1.100 - - [25/Dec/2024:10:17:20 +0000] "GET /contact.html HTTP/1.1"
404 196
```

Данный проект демонстрирует объединение всех изученных концепций: работу с текстовыми файлами через контекстные менеджеры, обработку исключений, использование JSON для конфигурации, CSV для отчетов, а также применение всех структур данных Python для эффективного анализа информации.

Проект легко расширяется: можно добавить анализ географии по IP, детекцию ботов по User-Agent, анализ трендов по дням недели, интеграцию с базами данных для хранения исторической статистики. Такой подход показывает, как из простых инструментов можно создавать мощные и полезные программы.

Глава 8. Модули и пакеты

8.1. Организация кода: от скрипта к модулю

Представьте себе столяра, который решил изготовить шкаф. Он может взять одну огромную доску и вырезать из нее весь шкаф целиком — это будет работать, но такой подход крайне неэффективен и неудобен. Вместо этого опытный мастер разделит задачу на компоненты: полки, дверцы, боковые стенки, каждая из которых изготавливается отдельно, а затем собирается в готовое изделие.

Аналогично обстоит дело с программированием. По мере роста вашего опыта и усложнения решаемых задач вы неизбежно столкнетесь с ситуацией, когда весь код в одном файле становится неуправляемым. Именно здесь на помощь приходит концепция модулей (modules) — фундаментальный принцип организации кода в Python.

Эволюция от монолитного скрипта

Давайте проследим естественную эволюцию программы от простого скрипта к модульной архитектуре. Представим, что мы разрабатываем систему управления библиотечными книгами. Начальная версия может выглядеть следующим образом:

```
# library_system.py - монолитная версия
# Данные о книгах
books = []
# Функция добавления книги
def add_book(title, author, year):
     book = {"title": title, "author": author, "year": year, "available":
True}
    books.append(book)
    print(f"Книга '{title}' добавлена в библиотеку")
# Функция поиска книги
def find_book(title):
    for book in books:
        if book["title"].lower() == title.lower():
            return book
    return None
# Функция выдачи книги
def lend_book(title):
    book = find_book(title)
    if book and book["available"]:
        book["available"] = False
        print(f"Книга '{title}' выдана читателю")
        return True
    else:
        print(f"Книга '{title}' недоступна")
        return False
# Функция возврата книги
def return_book(title):
```

```
    book = find_book(title)
    if book and not book["available"]:
        book["available"] = True
        print(f"Книга '{title}' возвращена")
        return True
    else:
        print(f"Книга '{title}' не была выдана")
        return False
# Функция отображения всех книг
def display_books():
    if not books:
        print("Библиотека пуста")
        return

    for book in books:
        status = "Доступна" if book["available"] else "Выдана"
          print(f"'{book['title']}' - {book['author']} ({book['year']}) -
{status}")
# Функция сохранения в файл
def save_to_file(filename):
    import json
    with open(filename, 'w', encoding='utf-8') as f:
        json.dump(books, f, ensure_ascii=False, indent=2)
    print(f"Данные сохранены в файл {filename}")
# Функция загрузки из файла
def load_from_file(filename):
    import json
    try:
        with open(filename, 'r', encoding='utf-8') as f:
            global books
            books = json.load(f)
        print(f"Данные загружены из файла {filename}")
    except FileNotFoundError:
        print(f"Файл {filename} не найден")
# Основная программа
if __name__ == "__main__":
    load_from_file("library.json")

    while True:
        print("\n=== Система управления библиотекой ===")
        print("1. Добавить книгу")
        print("2. Найти книгу")
        print("3. Выдать книгу")
        print("4. Вернуть книгу")
        print("5. Показать все книги")
        print("6. Сохранить и выйти")

        choice = input("Выберите действие: ")

        if choice == "1":
            title = input("Название: ")
```

```
            author = input("Автор: ")
            year = int(input("Год издания: "))
            add_book(title, author, year)
        elif choice == "2":
            title = input("Название для поиска: ")
            book = find_book(title)
            if book:
                status = "Доступна" if book["available"] else "Выдана"
                    print(f"Найдена: '{book['title']}' - {book['author']}
({book['year']}) - {status}")
            else:
                print("Книга не найдена")
        elif choice == "3":
            title = input("Название книги для выдачи: ")
            lend_book(title)
        elif choice == "4":
            title = input("Название возвращаемой книги: ")
            return_book(title)
        elif choice == "5":
            display_books()
        elif choice == "6":
            save_to_file("library.json")
            break
        else:
            print("Неверный выбор")
```

Эта программа функциональна, но по мере ее роста становятся очевидными серьезные недостатки монолитного подхода.

- **Проблема масштабируемости.** Представьте, что система должна обрабатывать не только книги, но и журналы, аудиокниги, электронные издания. Код станет огромным и трудночитаемым.
- **Отсутствие повторного использования.** Функции работы с книгами нельзя использовать в других проектах без копирования всего файла.
- **Сложность тестирования.** Проверить корректность отдельных функций затруднительно, поскольку они тесно связаны с глобальным состоянием.
- **Конфликты имен.** В большой программе легко случайно создать переменные или функции с одинаковыми именами.

Принципы модульности

Модульность (modularity) в программировании основывается на нескольких фундаментальных принципах, которые делают код более управляемым и надежным.

Принцип единственной ответственности. Каждый модуль должен решать одну конкретную задачу, и решать ее хорошо. В нашем примере логично выделить отдельные модули для работы с данными, файловых операций и пользовательского интерфейса.

Принцип слабой связанности. Модули должны минимально зависеть друг от друга. Изменения в одном модуле не должны требовать изменений в других модулях.

Принцип сильной связности. Элементы внутри одного модуля должны быть тесно связаны и работать сообща для достижения общей цели.

Принцип инкапсуляции. Модуль должен скрывать свои внутренние детали реализации и предоставлять четко определенный интерфейс для взаимодействия.

Преимущества модульной организации

Сравнивая модульную версию с монолитной, мы можем отметить существенные преимущества.

- **Читаемость и понятность.** Каждый модуль имеет четко определенную область ответственности. Разработчик может сосредоточиться на одном аспекте системы, не отвлекаясь на детали других компонентов.
- **Повторное использование.** Модуль `book_storage.py` может быть использован в других проектах, связанных с управлением библиотеками, без какой-либо модификации.
- **Простота тестирования.** Каждый модуль можно тестировать независимо от других. Например, можно легко проверить корректность операций с книгами без необходимости имитировать пользовательский интерфейс.
- **Параллельная разработка.** Разные разработчики могут работать над разными модулями одновременно, не мешая друг другу.
- **Упрощение отладки.** При возникновении ошибки легче локализовать проблему в конкретном модуле.

Модульность — это не просто способ организации кода, это мышление разработчика, которое позволяет создавать масштабируемые, поддерживаемые и надежные программы. По мере вашего профессионального роста вы будете все чаще применять этот принцип, создавая системы из множества взаимодействующих компонентов.

8.2. Импорт модулей: различные способы

Создание модулей — это только половина дела. Вторая, не менее важная, часть — это умение правильно импортировать и использовать функциональность других модулей. Python предоставляет несколько способов импорта, каждый из которых имеет свои особенности, преимущества и области применения.

Основы импорта: оператор import

Самый базовый способ импорта — использование оператора `import` с полным именем модуля:

```
import book_storage
import file_operations
import math
# Использование функций через имя модуля
book = book_storage.add_book("Война и мир", "Толстой Л.Н.", 1869)
data = file_operations.load_books_from_file("library.json")
result = math.sqrt(16)
```

Этот подход имеет важные преимущества. Он обеспечивает ясность происхождения каждой функции — всегда понятно, из какого модуля она взята. Кроме того, пол-

ностью исключается возможность конфликтов имен, поскольку функции разных модулей находятся в разных пространствах имен (namespaces).

Рассмотрим практический пример с модулем математических операций:

```
# math_operations.py
def factorial(n):
    """Вычисляет факториал числа."""
    if n <= 1:
        return 1
    return n * factorial(n - 1)
def fibonacci(n):
    """Возвращает n-й элемент последовательности Фибоначчи."""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
def is_prime(n):
    """Проверяет, является ли число простым."""
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
# main_calculator.py
import math_operations
print("Факториал 5:", math_operations.factorial(5))
print("10-й элемент Фибоначчи:", math_operations.fibonacci(10))
print("Число 17 простое:", math_operations.is_prime(17))
```

Селективный импорт: `from...import`

Когда из модуля нужны только определенные функции или константы, удобно использовать конструкцию `from...import`:

```
from math_operations import factorial, is_prime
from math import pi, sqrt, sin
# Теперь функции можно использовать напрямую
result = factorial(5)        # Вместо math_operations.factorial(5)
circle_area = pi * sqrt(25)  # Вместо math.pi * math.sqrt(25)
```

Этот подход делает код более лаконичным, особенно когда определенные функции используются часто. Однако он требует осторожности в отношении конфликтов имен:

```
from math import sqrt
from numpy import sqrt  # Ошибка! Перезапись функции sqrt
# Безопасный вариант:
from math import sqrt as math_sqrt
from numpy import sqrt as numpy_sqrt
```

Рассмотрим практический пример создания утилит для работы со строками:

```
# string_utils.py
def capitalize_words(text):
    """Делает заглавной первую букву каждого слова."""
    return ' '.join(word.capitalize() for word in text.split())
def remove_extra_spaces(text):
    """Удаляет лишние пробелы из строки."""
    return ' '.join(text.split())
def count_vowels(text):
    """Подсчитывает количество гласных в тексте."""
    vowels = "аеёиоуыэюяaeiouy"
    return sum(1 for char in text.lower() if char in vowels)
def reverse_words(text):
    """Переворачивает порядок слов в строке."""
    return ' '.join(reversed(text.split()))
# text_processor.py
from string_utils import capitalize_words, count_vowels
text = "привет мир программирования"
formatted_text = capitalize_words(text)
vowel_count = count_vowels(text)
print(f"Форматированный текст: {formatted_text}")
print(f"Количество гласных: {vowel_count}")
```

Переименование при импорте: оператор as

Иногда имена модулей или функций могут быть длинными или конфликтовать с существующими именами. В таких случаях используется оператор `as` для создания псевдонимов (aliases):

```
import book_storage as books
import file_operations as file_ops
from math_operations import factorial as fact
# Использование с псевдонимами
book = books.add_book("1984", "Оруэлл Дж.", 1949)
data = file_ops.load_books_from_file("data.json")
result = fact(6)
```

Псевдонимы особенно полезны при работе с модулями, имеющими длинные имена, или стандартными сокращениями в сообществе:

```
import datetime as dt
import json as js
import sqlite3 as db
current_time = dt.datetime.now()
data = js.loads('{"name": "Python", "version": 3.9}')
connection = db.connect("database.db")
```

Импорт всего содержимого: `from module import *`

Python позволяет импортировать все общедоступные имена из модуля с помощью конструкции `from module import *`:

```
from math import *
result = sqrt(16) + sin(pi/2) + log(e)
```

Однако этот подход настоятельно не рекомендуется в большинстве случаев по нескольким причинам.

- **Загрязнение пространства имен.** Все имена из модуля попадают в текущее пространство имен, что может привести к неожиданным конфликтам.
- **Снижение читаемости.** Становится неясно, откуда берется конкретная функция или переменная.
- **Проблемы при рефакторинге.** Сложно отследить зависимости и понять, какие именно функции используются.

Допустимые исключения включают интерактивную работу в Python REPL для быстрых экспериментов и импорт из модулей, специально предназначенных для такого использования (например, некоторые модули тестирования).

Абсолютные и относительные импорты

По мере усложнения структуры проекта возникает необходимость в организации модулей в пакеты (packages) — директории, содержащие несколько связанных модулей. В таких случаях важно понимать разницу между абсолютными и относительными импортами.

Рассмотрим структуру проекта библиотечной системы:

```
library_project/
├── main.py
├── storage/
│   ├── __init__.py
│   ├── book_storage.py
│   └── user_storage.py
├── utils/
│   ├── __init__.py
│   ├── file_operations.py
│   └── validation.py
└── ui/
    ├── __init__.py
    └── console_interface.py
```

Абсолютные импорты указывают полный путь к модулю от корня проекта:

```
# main.py
from storage.book_storage import add_book, find_book
from utils.file_operations import save_books_to_file
from ui.console_interface import display_menu
```

Относительные импорты используют точечную нотацию для указания относительного расположения модулей:

```
# storage/user_storage.py
from .book_storage import find_book  # Импорт из того же пакета
from ..utils.validation import validate_email  # Импорт из родительского пакета
```

Абсолютные импорты предпочтительны в большинстве случаев, поскольку они более явные и менее подвержены ошибкам при реструктуризации проекта.

Условный импорт и обработка ошибок

Иногда необходимо импортировать модули условно, в зависимости от их доступности или определенных условий:

```
# Попытка импорта с fallback
try:
    import ujson as json  # Более быстрая альтернатива
except ImportError:
    import json  # Стандартная библиотека
# Условный импорт в зависимости от платформы
import sys
if sys.platform == "win32":
    from utils.windows_specific import get_system_info
elif sys.platform.startswith("linux"):
    from utils.linux_specific import get_system_info
else:
    from utils.generic import get_system_info
```

Этот подход позволяет создавать кроссплатформенный код и использовать оптимальные библиотеки, когда они доступны.

Отложенный импорт

В некоторых случаях импорт модуля может быть дорогостоящей операцией, и его стоит выполнять только при необходимости:

```
def advanced_calculation(data):
    """Функция, использующая тяжелые математические библиотеки."""
    import numpy as np  # Импорт только при вызове функции
    import scipy.optimize

    # Сложные вычисления...
    return scipy.optimize.minimize(lambda x: np.sum(x**2), data)
def generate_report():
    """Генерация отчета с использованием библиотеки для работы с Excel."""
    try:
        import openpyxl  # Импорт только при необходимости
        # Создание Excel-отчета...
    except ImportError:
        print("Для создания Excel-отчетов установите библиотеку openpyxl")
        # Создание альтернативного формата отчета...
```

Практический пример: система логирования

Рассмотрим создание простой системы логирования, демонстрирующей различные способы импорта:

```
# logger.py
import datetime as dt
from pathlib import Path
from typing import Optional
```

```
class SimpleLogger:
    """Простая система логирования."""

    def __init__(self, log_file: Optional[str] = None):
        self.log_file = Path(log_file) if log_file else None

    def _format_message(self, level: str, message: str) -> str:
        timestamp = dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        return f"[{timestamp}] {level}: {message}"

    def info(self, message: str):
        formatted = self._format_message("INFO", message)
        print(formatted)
        if self.log_file:
            self._write_to_file(formatted)

    def error(self, message: str):
        formatted = self._format_message("ERROR", message)
        print(formatted)
        if self.log_file:
            self._write_to_file(formatted)

    def _write_to_file(self, message: str):
        with open(self.log_file, 'a', encoding='utf-8') as f:
            f.write(message + '\n')
# application.py
from logger import SimpleLogger
from storage.book_storage import add_book, find_book
from utils.file_operations import save_books_to_file as save_books
# Создаем экземпляр логгера
log = SimpleLogger("application.log")
def main():
    log.info("Приложение запущено")

    try:
        # Основная логика приложения
        book = add_book("Мастер и Маргарита", "Булгаков М.А.", 1967)
        log.info(f"Добавлена книга: {book['title']}")

        # Сохранение данных
        if save_books([book], "books.json"):
            log.info("Данные успешно сохранены")

    except Exception as e:
        log.error(f"Произошла ошибка: {e}")

    finally:
        log.info("Приложение завершено")
if __name__ == "__main__":
    main()
```

Лучшие практики импорта

При работе с импортами следуйте этим рекомендациям.

Порядок импортов. Стандартная практика — группировать импорты в следующем порядке: стандартная библиотека, сторонние пакеты, локальные модули. Группы разделяются пустой строкой:

```
# Стандартная библиотека
import os
import sys
import json
from pathlib import Path
# Сторонние пакеты
import requests
import numpy as np
import pandas as pd
# Локальные модули
from storage import book_storage
from utils.validation import validate_input
from .config import settings
```

Избегайте циклических импортов. Ситуация, когда модуль A импортирует модуль B, а модуль B импортирует модуль A, приводит к ошибкам. Решение — реструктуризация кода или использование отложенного импорта:

```
# Проблематично:
# module_a.py
from module_b import function_b
def function_a():
    return function_b() + 1
# module_b.py
from module_a import function_a  # Циклический импорт!
def function_b():
    return 42
# Решение – отложенный импорт:
# module_b.py
def function_b():
    from module_a import function_a  # Импорт внутри функции
    return 42
```

Используйте явные импорты. Предпочитайте конкретные импорты неопределенным:

```
# Хорошо
from math import sqrt, pi, sin
# Плохо
from math import *
```

Группируйте связанные импорты. При импорте множества элементов из одного модуля используйте скобки для лучшей читаемости:

```
from storage.book_storage import (
    add_book,
```

```
    find_book,
    lend_book,
    return_book,
    get_all_books
)
```

Документируйте нестандартные импорты. Если используете необычный способ импорта, объясните причину:

```
# Импортируем ujson для повышения производительности JSON-операций
try:
    import ujson as json
except ImportError:
    # Fallback на стандартную библиотеку
    import json
```

8.3. Стандартная библиотека Python: обзор возможностей

Одна из главных сильных сторон Python заключается в богатстве его стандартной библиотеки — обширной коллекции модулей, которые поставляются вместе с интерпретатором. Философия Python гласит: «Batteries included» (батарейки включены), и это означает, что для решения большинства типичных задач программирования вам не потребуется устанавливать дополнительные пакеты.

Стандартная библиотека Python содержит более 200 модулей, охватывающих практически все области программирования — от работы с операционной системой до сетевого программирования, от обработки данных до создания графических интерфейсов. Понимание возможностей стандартной библиотеки — это ключ к эффективной разработке на Python.

Модуль os: взаимодействие с операционной системой

Модуль `os` предоставляет переносимый интерфейс для работы с операционной системой. Он позволяет выполнять операции с файлами и каталогами, управлять процессами и получать системную информацию, не завися от конкретной операционной системы.

Основные возможности модуля `os`:

```
import os
# Получение текущего рабочего каталога
current_dir = os.getcwd()
print(f"Текущий каталог: {current_dir}")
# Список файлов и папок в каталоге
files = os.listdir('.')
print(f"Содержимое каталога: {files}")
# Создание новой папки
if not os.path.exists('temp_folder'):
    os.mkdir('temp_folder')
    print("Папка создана")
# Работа с путями
```

```
file_path = os.path.join('temp_folder', 'example.txt')
print(f"Полный путь: {file_path}")
# Проверка существования файла или папки
if os.path.exists(file_path):
    print("Файл существует")
else:
    print("Файл не найден")
```

Модуль `os.path` заслуживает особого внимания, поскольку предоставляет кросс-платформенные функции для работы с путями:

```
import os
# Разделение пути на компоненты
path = "/home/user/documents/file.txt"
directory, filename = os.path.split(path)
name, extension = os.path.splitext(filename)
print(f"Каталог: {directory}")
print(f"Имя файла: {name}")
print(f"Расширение: {extension}")
# Получение абсолютного пути
relative_path = "../data/input.csv"
absolute_path = os.path.abspath(relative_path)
print(f"Абсолютный путь: {absolute_path}")
```

Практический пример — функция для безопасного создания каталога:

```
import os
def create_directory_safe(path):
    """
    Создает каталог, включая промежуточные каталоги.
    Не выдает ошибку, если каталог уже существует.
    """
    try:
        os.makedirs(path, exist_ok=True)
        print(f"Каталог {path} готов к использованию")
        return True
    except OSError as e:
        print(f"Ошибка при создании каталога: {e}")
        return False
# Использование
create_directory_safe("reports/2024/january")
```

Модуль sys: информация о системе и интерпретаторе

Модуль `sys` предоставляет доступ к переменным и функциям, которые тесно связаны с интерпретатором Python. Он особенно полезен для получения системной информации и управления поведением программы:

```
import sys
# Информация о версии Python
print(f"Версия Python: {sys.version}")
```

```
print(f"Версия (краткая): {sys.version_info}")
# Пути поиска модулей
print("Пути поиска модулей:")
for path in sys.path:
    print(f"  {path}")
# Аргументы командной строки
print(f"Аргументы программы: {sys.argv}")
# Платформа
print(f"Платформа: {sys.platform}")
Практическое применение – создание универсального скрипта с параметрами:
import sys
def main():
    if len(sys.argv) < 2:
        print("Использование: python script.py <filename>")
        sys.exit(1)

    filename = sys.argv[1]

    try:
        with open(filename, 'r', encoding='utf-8') as f:
            content = f.read()
            print(f"Файл {filename} содержит {len(content)} символов")
    except FileNotFoundError:
        print(f"Ошибка: файл {filename} не найден")
        sys.exit(2)
    except Exception as e:
        print(f"Ошибка при чтении файла: {e}")
        sys.exit(3)
if __name__ == "__main__":
    main()
```

Модуль `datetime`: работа с датой и временем

Работа с датой и временем — одна из самых частых задач в программировании. Модуль `datetime` предоставляет классы для работы с датами, временем и временными интервалами.

```
from datetime import datetime, date, time, timedelta
# Текущие дата и время
now = datetime.now()
print(f"Сейчас: {now}")
today = date.today()
print(f"Сегодня: {today}")
# Создание конкретной даты
birthday = date(1990, 5, 15)
print(f"День рождения: {birthday}")
# Форматирование даты
formatted_date = now.strftime("%d.%m.%Y %H:%M:%S")
print(f"Отформатированная дата: {formatted_date}")
# Парсинг строки в дату
date_string = "2024-03-15 14:30:00"
```

```
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print(f"Распознанная дата: {parsed_date}")
```

Работа с временными интервалами:

```
from datetime import datetime, timedelta
# Временные интервалы
week_ago = datetime.now() - timedelta(days=7)
tomorrow = datetime.now() + timedelta(days=1)
print(f"Неделю назад: {week_ago}")
print(f"Завтра: {tomorrow}")
# Вычисление возраста
def calculate_age(birth_date):
    today = date.today()
    age = today.year - birth_date.year
    # Проверяем, был ли уже день рождения в этом году
    if today.month < birth_date.month or \
       (today.month == birth_date.month and today.day < birth_date.day):
        age -= 1
    return age
birth = date(1995, 8, 20)
age = calculate_age(birth)
print(f"Возраст: {age} лет")
```

Модули `math` и `random`: математические вычисления

Модуль `math` предоставляет математические функции и константы, определенные стандартом C:

```
import math
# Математические константы
print(f"Число π: {math.pi}")
print(f"Число e: {math.e}")
# Тригонометрические функции
angle = math.pi / 4  # 45 градусов в радианах
print(f"sin(45°) = {math.sin(angle)}")
print(f"cos(45°) = {math.cos(angle)}")
# Логарифмы и степени
print(f"log₁₀(100) = {math.log10(100)}")
print(f"2³ = {math.pow(2, 3)}")
print(f"√16 = {math.sqrt(16)}")
# Округление и деление
print(f"Потолок 3.2: {math.ceil(3.2)}")
print(f"Пол 3.8: {math.floor(3.8)}")
Модуль random генерирует псевдослучайные числа:
import random
# Случайное число от 0 до 1
print(f"Случайное число: {random.random()}")
# Случайное целое число в диапазоне
dice_roll = random.randint(1, 6)
```

```
print(f"Бросок кубика: {dice_roll}")
# Случайный выбор из списка
colors = ['красный', 'синий', 'зеленый', 'желтый']
chosen_color = random.choice(colors)
print(f"Выбранный цвет: {chosen_color}")
# Перемешивание списка
cards = list(range(1, 11))
random.shuffle(cards)
print(f"Перемешанные карты: {cards}")
```

Практический пример — генератор паролей:

```
import random
import string
def generate_password(length=12, include_symbols=True):
    """Генерирует случайный пароль заданной длины"""
    characters = string.ascii_letters + string.digits
    if include_symbols:
        characters += "!@#$%^&*"

    password = ''.join(random.choice(characters) for _ in range(length))
    return password
# Генерация нескольких паролей
for i in range(3):
    password = generate_password(16)
    print(f"Пароль {i+1}: {password}")
```

Модуль json: работа с данными JSON

JSON (JavaScript Object Notation) стал стандартом для обмена данными в веб-приложениях и API. Модуль `json` позволяет легко конвертировать данные Python в JSON и обратно:

```
import json
# Данные Python
data = {
    "name": "Анна Иванова",
    "age": 28,
    "city": "Москва",
    "languages": ["Python", "JavaScript", "SQL"],
    "is_programmer": True
}
# Конвертация в JSON-строку
json_string = json.dumps(data, ensure_ascii=False, indent=2)
print("JSON-строка:")
print(json_string)
# Конвертация JSON-строки обратно в объект Python
parsed_data = json.loads(json_string)
print(f"\nИмя: {parsed_data['name']}")
print(f"Языки программирования: {', '.join(parsed_data['languages'])}")
```

Работа с JSON-файлами:

```
import json
# Сохранение данных в файл
user_data = {
    "users": [
        {"id": 1, "name": "Петр", "email": "peter@example.com"},
        {"id": 2, "name": "Мария", "email": "maria@example.com"}
    ]
}
# Запись в файл
with open('users.json', 'w', encoding='utf-8') as f:
    json.dump(user_data, f, ensure_ascii=False, indent=4)
# Чтение из файла
with open('users.json', 'r', encoding='utf-8') as f:
    loaded_data = json.load(f)
print("Загруженные пользователи:")
for user in loaded_data['users']:
    print(f"  {user['name']} - {user['email']}")
```

Интеграция модулей: комплексный пример

Рассмотрим практический пример, который объединяет несколько изученных модулей — систему логирования событий:

```
import os
import json
import sys
from datetime import datetime
class EventLogger:
    def __init__(self, log_directory="logs"):
        self.log_directory = log_directory
        self.ensure_log_directory()

    def ensure_log_directory(self):
        """Создает директорию для логов, если она не существует"""
        if not os.path.exists(self.log_directory):
            os.makedirs(self.log_directory)
            print(f"Создана директория логов: {self.log_directory}")

    def log_event(self, event_type, message, additional_data=None):
        """Записывает событие в лог-файл"""
        timestamp = datetime.now()

        log_entry = {
            "timestamp": timestamp.isoformat(),
            "type": event_type,
            "message": message,
            "python_version": sys.version.split()[0],
            "additional_data": additional_data or {}
        }
```

```
        # Имя файла на основе даты
        log_filename = f"log_{timestamp.strftime('%Y_%m_%d')}.json"
        log_path = os.path.join(self.log_directory, log_filename)
                # Чтение существующих записей или создание нового списка
        if os.path.exists(log_path):
            with open(log_path, 'r', encoding='utf-8') as f:
                logs = json.load(f)
        else:
            logs = []

        # Добавление новой записи
        logs.append(log_entry)

        # Сохранение обратно в файл
        with open(log_path, 'w', encoding='utf-8') as f:
            json.dump(logs, f, ensure_ascii=False, indent=2)

        print(f"Событие записано: {event_type} - {message}")
# Использование системы логирования
logger = EventLogger()
logger.log_event("INFO", "Программа запущена")
logger.log_event("ERROR", "Файл не найден", {"filename": "data.csv"})
logger.log_event("SUCCESS", "Данные успешно обработаны", {"records_count":
1500})
```

8.4. Создание собственных пакетов

По мере роста ваших проектов возникает необходимость в организации кода в более структурированном виде. Пакеты (packages) в Python — это способ группировки связанных модулей в иерархические структуры, подобные папкам в файловой системе. Понимание принципов создания пакетов критически важно для разработки масштабируемых и поддерживаемых приложений.

Анатомия пакета: структура и принципы

Пакет в Python — это директория, содержащая файл `__init__.py` (может быть пустым) и один или несколько модулей. Этот файл сообщает интерпретатору Python, что данная директория должна рассматриваться как пакет.

Рассмотрим структуру типичного пакета для работы с математическими вычислениями:

```
mathtools/
├── __init__.py
├── basic.py
├── statistics.py
├── geometry/
│   ├── __init__.py
│   ├── shapes.py
│   └── calculations.py
└── utils/
    ├── __init__.py
    └── helpers.py
```

Создание базового пакета

Начнем с создания простого пакета `mathtools`. Создайте директорию с таким именем и добавьте в нее файлы:

```
Файл mathtools/__init__.py:
"""
MathTools - пакет для математических вычислений
Содержит модули для базовых операций, статистики и геометрии.
"""
__version__ = "1.0.0"
__author__ = "Ваше Имя"
# Импорты для удобного доступа к функциям пакета
from .basic import add, multiply, power
from .statistics import mean, median
# Список функций, доступных при импорте *
__all__ = ['add', 'multiply', 'power', 'mean', 'median']
```

Файл `mathtools/basic.py`:

```
"""Базовые математические операции"""
def add(a, b):
    """Сложение двух чисел"""
    return a + b
def subtract(a, b):
    """Вычитание второго числа из первого"""
    return a - b
def multiply(a, b):
    """Умножение двух чисел"""
    return a * b
def divide(a, b):
    """Деление первого числа на второе"""
    if b == 0:
        raise ValueError("Деление на ноль невозможно")
    return a / b
def power(base, exponent):
    """Возведение числа в степень"""
    return base ** exponent
```

Файл `mathtools/statistics.py`:

```
"""Функции для статистических вычислений"""
def mean(numbers):
    """Вычисляет среднее арифметическое списка чисел"""
    if not numbers:
        raise ValueError("Список не может быть пустым")
    return sum(numbers) / len(numbers)
def median(numbers):
    """Вычисляет медиану списка чисел"""
    if not numbers:
        raise ValueError("Список не может быть пустым")

    sorted_numbers = sorted(numbers)
```

```
    n = len(sorted_numbers)

    if n % 2 == 1:
        # Нечетное количество элементов
        return sorted_numbers[n // 2]
    else:
        # Четное количество элементов
        middle1 = sorted_numbers[n // 2 - 1]
        middle2 = sorted_numbers[n // 2]
        return (middle1 + middle2) / 2
def mode(numbers):
    """Вычисляет моду (наиболее часто встречающееся значение)"""
    if not numbers:
        raise ValueError("Список не может быть пустым")

    frequency = {}
    for number in numbers:
        frequency[number] = frequency.get(number, 0) + 1

    max_frequency = max(frequency.values())
   modes = [num for num, freq in frequency.items() if freq == max_frequency]

    return modes[0] if len(modes) == 1 else modes
```

Создание вложенных пакетов

Теперь создадим подпакет `geometry` для геометрических вычислений:

Файл `mathtools/geometry/__init__.py`:

```
"""Модуль геометрических вычислений"""
from .shapes import Circle, Rectangle
from .calculations import area_circle, area_rectangle, perimeter_circle,
perimeter_rectangle
__all__ = ['Circle', 'Rectangle', 'area_circle', 'area_rectangle',
           'perimeter_circle', 'perimeter_rectangle']
```

Файл `mathtools/geometry/shapes.py`:

```
"""Классы геометрических фигур"""
import math
class Circle:
    """Класс для работы с кругом"""

    def __init__(self, radius):
        if radius <= 0:
            raise ValueError("Радиус должен быть положительным числом")
        self.radius = radius

    def area(self):
        """Вычисляет площадь круга"""
        return math.pi * self.radius ** 2
```

```
        def perimeter(self):
        """Вычисляет длину окружности"""
        return 2 * math.pi * self.radius
class Rectangle:
    """Класс для работы с прямоугольником"""

    def __init__(self, width, height):
        if width <= 0 or height <= 0:
              raise ValueError("Ширина и высота должны быть положительными
числами")
        self.width = width
        self.height = height

    def area(self):
        """Вычисляет площадь прямоугольника"""
        return self.width * self.height

    def perimeter(self):
        """Вычисляет периметр прямоугольника"""
        return 2 * (self.width + self.height)
```

Файл `mathtools/geometry/calculations.py`:

```
"""Функции для геометрических вычислений"""
import math
def area_circle(radius):
    """Вычисляет площадь круга по радиусу"""
    if radius <= 0:
        raise ValueError("Радиус должен быть положительным числом")
    return math.pi * radius ** 2
def area_rectangle(width, height):
    """Вычисляет площадь прямоугольника"""
    if width <= 0 or height <= 0:
         raise ValueError("Ширина и высота должны быть положительными чис-
лами")
    return width * height
def perimeter_circle(radius):
    """Вычисляет длину окружности"""
    if radius <= 0:
        raise ValueError("Радиус должен быть положительным числом")
    return 2 * math.pi * radius
def perimeter_rectangle(width, height):
    """Вычисляет периметр прямоугольника"""
    if width <= 0 or height <= 0:
         raise ValueError("Ширина и высота должны быть положительными чис-
лами")
    return 2 * (width + height)
```

Использование созданного пакета

После создания структуры пакета вы можете импортировать и использовать его различными способами:

```
# Импорт всего пакета
import mathtools
# Использование функций через пакет
result1 = mathtools.add(10, 5)
result2 = mathtools.mean([1, 2, 3, 4, 5])
print(f"Сложение: {result1}")
print(f"Среднее значение: {result2}")
# Импорт конкретных функций
from mathtools import multiply, median
from mathtools.statistics import mode
numbers = [1, 2, 2, 3, 4, 4, 4, 5]
print(f"Произведение: {multiply(6, 7)}")
print(f"Медиана: {median(numbers)}")
print(f"Мода: {mode(numbers)}")
# Импорт подпакета
from mathtools.geometry import Circle, area_rectangle
circle = Circle(5)
print(f"Площадь круга: {circle.area():.2f}")
print(f"Площадь прямоугольника: {area_rectangle(4, 6)}")
# Импорт с псевдонимом
import mathtools.geometry.calculations as geom_calc
area = geom_calc.area_circle(3)
print(f"Площадь круга (через псевдоним): {area:.2f}")
```

Относительные импорты внутри пакета

Внутри пакета модули могут импортировать друг друга, используя относительные импорты. Это улучшает переносимость и ясность кода:

Создадим файл `mathtools/utils/helpers.py`:

```
"""Вспомогательные функции для пакета mathtools"""
def validate_positive_number(value, name="число"):
    """Проверяет, что значение является положительным числом"""
    if not isinstance(value, (int, float)):
        raise TypeError(f"{name} должно быть числом")
    if value <= 0:
        raise ValueError(f"{name} должно быть положительным")
    return True
def validate_non_empty_list(lst, name="список"):
    """Проверяет, что список не пустой"""
    if not isinstance(lst, list):
        raise TypeError(f"{name} должен быть списком")
    if not lst:
        raise ValueError(f"{name} не может быть пустым")
    return True
```

Теперь обновим `mathtools/statistics.py` для использования вспомогательных функций:

```
"""Функции для статистических вычислений"""
# Относительный импорт из подпакета utils
from .utils.helpers import validate_non_empty_list
def mean(numbers):
    """Вычисляет среднее арифметическое списка чисел"""
    validate_non_empty_list(numbers, "список чисел")
    return sum(numbers) / len(numbers)
def median(numbers):
    """Вычисляет медиану списка чисел"""
    validate_non_empty_list(numbers, "список чисел")

    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)

    if n % 2 == 1:
        return sorted_numbers[n // 2]
    else:
        middle1 = sorted_numbers[n // 2 - 1]
        middle2 = sorted_numbers[n // 2]
        return (middle1 + middle2) / 2
```

8.5. Менеджер пакетов `pip`: расширяем возможности

Изучив основы создания модулей и пакетов, мы переходим к одной из самых мощных особенностей экосистемы Python — системе управления пакетами. Представьте себе, что вам нужно реализовать сложные математические вычисления, работу с веб-API или анализ данных. Написание всего функционала с нуля потребовало бы месяцев работы, но благодаря богатой экосистеме Python вы можете воспользоваться готовыми решениями, созданными тысячами разработчиков по всему миру.

Философия повторного использования кода

Менеджер пакетов `pip` (Pip Installs Packages) воплощает в себе одну из фундаментальных идей программирования — принцип DRY (Don't Repeat Yourself), распространенный на уровень всего сообщества. Вместо того чтобы каждый разработчик изобретал велосипед заново, `pip` позволяет легко находить, устанавливать и использовать код, написанный другими программистами.

PyPI (Python Package Index) — это централизованный репозиторий, содержащий сотни тысяч пакетов для самых разных задач: от простых утилит до сложных фреймворков машинного обучения. Это делает Python невероятно продуктивным языком для разработки.

Основные операции с `pip`

Рассмотрим базовые команды для работы с `pip`. Все команды выполняются в терминале или командной строке.

Проверка версии pip и получение справки:

```
pip --version
pip help
pip help install  # Справка по конкретной команде
```

Поиск пакетов в PyPI:

```
pip search requests  # Поиск пакетов, связанных с HTTP-запросами
```

Установка пакетов — самая частая операция:

```
pip install requests            # Установка последней версии
pip install requests==2.28.0    # Установка конкретной версии
pip install "requests>=2.25.0"  # Установка с ограничением по версии
```

Просмотр установленных пакетов:

```
pip list                  # Все установленные пакеты
pip show requests         # Подробная информация о пакете
pip list --outdated       # Пакеты, для которых доступны обновления
```

Управление зависимостями проекта

В реальной разработке проект обычно использует десятки внешних пакетов. Управление этими зависимостями (dependencies) — критически важная задача. Основным инструментом для этого служит файл `requirements.txt`.

Создание файла зависимостей:

```
pip freeze > requirements.txt
```

Этот файл будет содержать список всех установленных пакетов с точными версиями:

```
certifi==2022.12.7
charset-normalizer==3.1.0
idna==3.4
requests==2.28.2
urllib3==1.26.15
```

Восстановление окружения из файла зависимостей:

```
pip install -r requirements.txt
```

Эта возможность делает проекты воспроизводимыми (reproducible) — другой разработчик может точно воссоздать ваше рабочее окружение.

Практический пример: работа с внешними библиотеками

Продемонстрируем практическое использование pip на примере создания программы для получения информации о погоде. Для этого установим популярную библиотеку `requests` для работы с HTTP:

```
pip install requests
```

Теперь создадим простой скрипт для получения данных о погоде:

```
import requests
import json
def get_weather_info(city):
    """
    Получает информацию о погоде для указанного города
    Использует бесплатный API OpenWeatherMap
    """
    # API-ключ нужно получить на сайте openweathermap.org
    api_key = "your_api_key_here"
    url = f"https://api.openweathermap.org/data/2.5/weather"

    params = {
        'q': city,
        'appid': api_key,
        'units': 'metric',  # Температура в градусах Цельсия
        'lang': 'ru'        # Описание на русском
    }

    try:
        response = requests.get(url, params=params)
        response.raise_for_status()  # Вызовет исключение при ошибке HTTP

        weather_data = response.json()
        return {
            'city': weather_data['name'],
            'temperature': weather_data['main']['temp'],
            'description': weather_data['weather'][0]['description'],
            'humidity': weather_data['main']['humidity']
        }

    except requests.RequestException as e:
        print(f"Ошибка при запросе данных: {e}")
        return None
# Пример использования
if __name__ == "__main__":
    city_name = input("Введите название города: ")
    weather = get_weather_info(city_name)

    if weather:
        print(f"Погода в {weather['city']}:")
        print(f"Температура: {weather['temperature']}°C")
        print(f"Описание: {weather['description']}")
        print(f"Влажность: {weather['humidity']}%")
```

Обновление и удаление пакетов

Поддержание пакетов в актуальном состоянии — важная часть управления проектом:

```
pip install --upgrade requests     # Обновление конкретного пакета
pip install --upgrade pip          # Обновление самого pip
```

```
pip uninstall requests              # Удаление пакета
pip uninstall -r requirements.txt  # Удаление всех пакетов из файла
```

Работа с индексами и источниками

По умолчанию pip использует официальный PyPI, но иногда нужны альтернативные источники:

```
# Установка из альтернативного индекса
pip install --index-url https://test.pypi.org/simple/ your-package
# Установка из git-репозитория
pip install git+https://github.com/user/repo.git
# Установка локального пакета в режиме разработки
pip install -e .  # Из текущей директории
```

Понимание pip открывает перед вами огромный мир готовых решений. Однако с большой силой приходит большая ответственность — важно научиться изолировать проекты друг от друга, чтобы зависимости одного не влияли на другой. Именно для решения этой задачи существуют виртуальные окружения, которые мы изучим в следующем разделе.

8.6. Виртуальные окружения: изоляция проектов

Представьте ситуацию: вы работаете над двумя проектами одновременно. Первый требует Django версии 3.2, а второй — Django версии 4.1. Как установить обе версии на одной машине? Или другой сценарий: вы хотите поэкспериментировать с новой версией библиотеки, но боитесь «сломать» рабочий проект. Именно для решения подобных проблем существуют виртуальные окружения (virtual environments).

Проблема глобального пространства

Когда вы устанавливаете пакеты с помощью pip без дополнительных настроек, они попадают в глобальное пространство Python. Это означает, что все проекты на вашей машине используют одни и те же версии библиотек. Проблемы начинаются, когда разные проекты требуют разных версий одного пакета или когда обновление библиотеки для одного проекта «ломает» другой.

Виртуальное окружение решает эту проблему, создавая изолированное пространство для каждого проекта. В каждом окружении может быть своя версия Python и свой набор установленных пакетов, которые не влияют на другие проекты.

venv: стандартный инструмент изоляции

Начиная с Python 3.3 в стандартную библиотеку входит модуль venv, предоставляющий базовую функциональность виртуальных окружений. Это рекомендуемый способ создания изолированных сред для большинства случаев.

Создание виртуального окружения:

```
# Создание окружения в папке myproject_env
python -m venv myproject_env
```

```
# В Windows также можно использовать:
py -m venv myproject_env
```

Эта команда создает директорию `myproject_env` со следующей структурой:

```
myproject_env/
├── bin/ (Scripts/ в Windows)     # Исполняемые файлы
├── include/                      # Заголовочные файлы C
├── lib/                         # Установленные пакеты
├── lib64/                       # Ссылка на lib (только Unix)
└── pyvenv.cfg                   # Конфигурация окружения
```

Активация виртуального окружения — ключевой момент в работе:

```
# В Unix/Linux/macOS:
source myproject_env/bin/activate
# В Windows:
myproject_env\Scripts\activate
```

После активации в командной строке появится префикс с именем окружения:

```
(myproject_env) user@computer:~/projects$
```

Это означает, что все операции с `pip` и Python будут происходить внутри изолированного окружения.

Альтернативы venv

virtualenv

`virtualenv` — более старый, но более мощный инструмент, который работает с Python 2 и предоставляет дополнительные возможности:

```
# Установка
pip install virtualenv
# Создание окружения
virtualenv myproject_env
# Создание с конкретной версией Python
virtualenv -p python3.9 myproject_env
```

conda

Для задач, связанных с анализом данных и машинным обучением, популярен менеджер пакетов `conda`, который управляет не только Python-пакетами, но и системными библиотеками:

```
# Создание окружения
conda create --name myproject python=3.9
# Активация
conda activate myproject
# Установка пакетов
conda install numpy pandas matplotlib
# Экспорт окружения
conda env export > environment.yml
# Создание окружения из файла
conda env create -f environment.yml
```

Лучшие практики работы с виртуальными окружениями

Одно окружение — один проект. Не переиспользуйте окружения между проектами, даже если они кажутся похожими. Это избавит от неожиданных конфликтов в будущем.

Исключение из системы контроля версий. Виртуальные окружения не должны попадать в git-репозиторий. Добавьте папки окружений в `.gitignore`:

```
# .gitignoreenv/venv/myproject_env/.conda/
```

Документирование зависимостей. Всегда ведите актуальный файл `requirements.txt` или `environment.yml`:

```
# Для pip
pip freeze > requirements.txt
# Для conda
conda env export --no-builds > environment.yml
```

Именование окружений. Используйте понятные имена, отражающие назначение проекта:

```
python -m venv data_analysis_env
python -m venv web_scraper_env
python -m venv django_blog_env
```

Автоматизация работы с окружениями

Для упрощения работы можно создать bash-скрипт (или batch-файл в Windows) для быстрого создания и настройки окружений:

```
#!/bin/bash
# setup_project.sh
PROJECT_NAME=$1
if [ -z "$PROJECT_NAME" ]; then
    echo "Использование: ./setup_project.sh <имя_проекта>"
    exit 1
fi
echo "Создание проекта $PROJECT_NAME..."
# Создаем директорию проекта
mkdir "$PROJECT_NAME"
cd "$PROJECT_NAME"
# Создаем виртуальное окружение
python -m venv env
# Активируем окружение
source env/bin/activate
# Обновляем pip
pip install --upgrade pip
# Создаём базовые файлы
touch main.py
touch requirements.txt
echo "env/" > .gitignore
echo "Проект $PROJECT_NAME готов к работе!"
```

```
echo "Для активации окружения выполните: source $PROJECT_NAME/env/bin/activate"
```

Виртуальные окружения — это не просто техническая особенность, это культура разработки, которая делает ваши проекты надежными, воспроизводимыми и профессиональными. Потратив немного времени на освоение этого инструмента, вы значительно упростите себе жизнь в долгосрочной перспективе и избежите множества проблем, связанных с конфликтами зависимостей.

8.7. Практическая работа: создание утилитарного пакета

В этом разделе мы пошагово создадим полноценный Python-пакет, который будет содержать утилиты для работы с текстом и файлами. Этот практический опыт покажет весь цикл разработки пакета — от планирования структуры до подготовки к публикации.

Планирование пакета

Прежде чем писать код, необходимо четко определить назначение и структуру нашего пакета. Создадим пакет `textutils` (утилиты для текста), который будет включать функции для:

- анализа текста (подсчет слов, символов, предложений);
- очистки и форматирования текста;
- работы с файлами (массовое чтение/запись текстовых файлов);
- простых статистических операций над текстом.

Такой пакет демонстрирует реальную практическую ценность — подобные утилиты часто нужны в различных проектах, от анализа данных до веб-разработки.

Создание структуры пакета

Начнем с создания правильной файловой структуры. В отдельной папке создадим следующие файлы и директории:

```
textutils/
├── textutils/
│   ├── __init__.py
│   ├── analyzer.py
│   ├── cleaner.py
│   └── fileops.py
├── setup.py
├── README.md
└── requirements.txt
```

Такая структура является стандартной для Python-пакетов. Внешняя папка `textutils` содержит весь проект, внутренняя папка с тем же именем — собственно код пакета.

Создание модулей пакета

Модуль анализа текста (`analyzer.py`)
Начнем с модуля для анализа текстовых данных:

```
"""
Модуль для анализа текстовых данных
"""
import re
from collections import Counter
def count_words(text):
    """
    Подсчитывает количество слов в тексте

    Args:
        text (str): Исходный текст для анализа

    Returns:
        int: Количество слов
    """
    if not text or not isinstance(text, str):
        return 0

    # Удаляем лишние пробелы и разбиваем по словам
    words = text.strip().split()
    return len(words)
def count_characters(text, include_spaces=True):
    """
    Подсчитывает количество символов в тексте

    Args:
        text (str): Исходный текст
        include_spaces (bool): Учитывать ли пробелы

    Returns:
        int: Количество символов
    """
    if not text or not isinstance(text, str):
        return 0

    if include_spaces:
        return len(text)
    else:
        return len(text.replace(' ', ''))
def count_sentences(text):
    """
    Подсчитывает количество предложений в тексте

    Args:
        text (str): Исходный текст

    Returns:
        int: Количество предложений
```

```
    """
    if not text or not isinstance(text, str):
        return 0

    # Простой подсчет по знакам препинания
    sentence_endings = re.findall(r'[.!?]+', text)
    return len(sentence_endings)
def word_frequency(text):
    """
    Возвращает частоту употребления слов

    Args:
        text (str): Исходный текст

    Returns:
        dict: Словарь {слово: частота}
    """
    if not text or not isinstance(text, str):
        return {}

    # Приводим к нижнему регистру и разбиваем
    words = text.lower().split()
    # Убираем знаки препинания
    clean_words = [re.sub(r'[^\w]', '', word) for word in words if word]
    # Считаем частоты
    return dict(Counter(clean_words))
def text_statistics(text):
    """
    Возвращает полную статистику по тексту

    Args:
        text (str): Исходный текст

    Returns:
        dict: Словарь со статистикой
    """
    if not text or not isinstance(text, str):
        return {
            'words': 0,
            'characters': 0,
            'characters_no_spaces': 0,
            'sentences': 0,
            'word_frequency': {}
        }

    return {
        'words': count_words(text),
        'characters': count_characters(text, True),
        'characters_no_spaces': count_characters(text, False),
        'sentences': count_sentences(text),
        'word_frequency': word_frequency(text)
    }
```

Модуль очистки текста (`cleaner.py`)

Следующий модуль будет содержать функции для очистки и форматирования текста:

```
"""
Модуль для очистки и форматирования текста
"""
import re
def remove_extra_spaces(text):
    """
    Удаляет лишние пробелы из текста

    Args:
        text (str): Исходный текст

    Returns:
        str: Очищенный текст
    """
    if not text or not isinstance(text, str):
        return ""

    # Заменяет множественные пробелы на одинарные
    cleaned = re.sub(r'\s+', ' ', text)
    return cleaned.strip()
def remove_punctuation(text, keep_periods=False):
    """
    Удаляет знаки препинания из текста

    Args:
        text (str): Исходный текст
        keep_periods (bool): Оставить ли точки

    Returns:
        str: Текст без знаков препинания
    """
    if not text or not isinstance(text, str):
        return ""

    if keep_periods:
        # Убираем все, кроме букв, цифр, пробелов и точек
        cleaned = re.sub(r'[^\w\s.]', '', text)
    else:
        # Убираем все, кроме букв, цифр и пробелов
        cleaned = re.sub(r'[^\w\s]', '', text)

    return remove_extra_spaces(cleaned)
def normalize_case(text, mode='lower'):
    """
    Нормализует регистр текста

    Args:
        text (str): Исходный текст
        mode (str): Режим ('lower', 'upper', 'title', 'sentence')
```

```
    Returns:
        str: Нормализованный текст
    """
    if not text or not isinstance(text, str):
        return ""

    if mode == 'lower':
        return text.lower()
    elif mode == 'upper':
        return text.upper()
    elif mode == 'title':
        return text.title()
    elif mode == 'sentence':
        # Первая буква заглавная, остальные строчные
        return text[0].upper() + text[1:].lower() if text else ""
    else:
        return text
def clean_text(text, remove_punct=True, normalize=True):
    """
    Комплексная очистка текста

    Args:
        text (str): Исходный текст
        remove_punct (bool): Удалить знаки препинания
        normalize (bool): Привести к нижнему регистру

    Returns:
        str: Очищенный текст
    """
    if not text or not isinstance(text, str):
        return ""

    cleaned = remove_extra_spaces(text)

    if remove_punct:
        cleaned = remove_punctuation(cleaned)

    if normalize:
        cleaned = normalize_case(cleaned, 'lower')

    return cleaned
```

Модуль работы с файлами (`fileops.py`)

Создадим модуль для работы с текстовыми файлами:

```
"""
Модуль для работы с текстовыми файлами
"""
import os
from .analyzer import text_statistics
from .cleaner import clean_text
```

```
def read_text_file(filepath, encoding='utf-8'):
    """
    Читает текстовый файл

    Args:
        filepath (str): Путь к файлу
        encoding (str): Кодировка файла

    Returns:
        str: Содержимое файла или None при ошибке
    """
    try:
        with open(filepath, 'r', encoding=encoding) as file:
            return file.read()
    except FileNotFoundError:
        print(f"Файл {filepath} не найден")
        return None
    except Exception as e:
        print(f"Ошибка при чтении файла {filepath}: {e}")
        return None
def write_text_file(filepath, content, encoding='utf-8'):
    """
    Записывает текст в файл

    Args:
        filepath (str): Путь к файлу
        content (str): Содержимое для записи
        encoding (str): Кодировка файла

    Returns:
        bool: True при успехе, False при ошибке
    """
    try:
        with open(filepath, 'w', encoding=encoding) as file:
            file.write(content)
        return True
    except Exception as e:
        print(f"Ошибка при записи файла {filepath}: {e}")
        return False
def process_text_files(directory, output_file=None):
    """
    Обрабатывает все текстовые файлы в директории

    Args:
        directory (str): Путь к директории
        output_file (str): Файл для сохранения результатов

    Returns:
        dict: Результаты анализа всех файлов
    """
    results = {}
```

```
    if not os.path.exists(directory):
        print(f"Директория {directory} не существует")
        return results

    for filename in os.listdir(directory):
        if filename.endswith('.txt'):
            filepath = os.path.join(directory, filename)
            content = read_text_file(filepath)

            if content:
                stats = text_statistics(content)
                results[filename] = stats

    if output_file and results:
        # Создаем отчет
        report = "Отчет по анализу текстовых файлов\n"
        report += "=" * 40 + "\n\n"

        for filename, stats in results.items():
            report += f"Файл: {filename}\n"
            report += f"Слов: {stats['words']}\n"
            report += f"Символов: {stats['characters']}\n"
            report += f"Предложений: {stats['sentences']}\n"
            report += "-" * 20 + "\n"

        write_text_file(output_file, report)

    return results
```

Файл инициализации пакета (`init.py`)

Создадим файл инициализации, который определяет публичный API пакета:

```
"""
TextUtils - утилиты для работы с текстом
Пакет предоставляет функции для:
- Анализа текста (подсчет слов, символов, статистика)
- Очистки и форматирования текста
- Работы с текстовыми файлами
"""
from .analyzer import (
    count_words,
    count_characters,
    count_sentences,
    word_frequency,
    text_statistics
)
from .cleaner import (
    remove_extra_spaces,
    remove_punctuation,
    normalize_case,
```

```
    clean_text
)
from .fileops import (
    read_text_file,
    write_text_file,
    process_text_files
)
__version__ = "1.0.0"
__author__ = "Your Name"
__email__ = "your.email@example.com"
# Определяем, что будет доступно при импорте *
__all__ = [
    'count_words',
    'count_characters',
    'count_sentences',
    'word_frequency',
    'text_statistics',
    'remove_extra_spaces',
    'remove_punctuation',
    'normalize_case',
    'clean_text',
    'read_text_file',
    'write_text_file',
    'process_text_files'
]
```

Создание файла установки (`setup.py`)

Файл `setup.py` определяет, как пакет должен быть установлен:

```
from setuptools import setup, find_packages
# Читаем содержимое README-файла
with open("README.md", "r", encoding="utf-8") as fh:
    long_description = fh.read()
setup(
    name="textutils",
    version="1.0.0",
    author="Your Name",
    author_email="your.email@example.com",
    description="Утилиты для работы с текстом на Python",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/yourusername/textutils",
    packages=find_packages(),
    classifiers=[
        "Development Status :: 4 - Beta",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
        "Programming Language :: Python :: 3",
        "Programming Language :: Python :: 3.7",
        "Programming Language :: Python :: 3.8",
```

```
        "Programming Language :: Python :: 3.9",
        "Programming Language :: Python :: 3.10",
    ],
    python_requires=">=3.7",
    install_requires=[
        # Наш пакет использует только стандартную библиотеку
    ],
    extras_require={
        "dev": [
            "pytest>=6.0",
            "black",
            "flake8",
        ],
    },
)
```

Создание документации (README.md)
Создадим простую документацию для пакета:

```
# TextUtils
Удобные утилиты для работы с текстом на Python.
## Установка
```bash
pip install textutils
```

Использование

Анализ текста

```
from textutils import text_statistics, count_words
text = "Пример текста для анализа. Содержит несколько предложений!"
stats = text_statistics(text)
print(f"Слов: {stats['words']}")
```

Очистка текста

```
from textutils import clean_text
dirty_text = "  Текст   с лишними    пробелами!!! "
clean = clean_text(dirty_text)
print(clean)  # "текст с лишними пробелами"
```

Работа с файлами

```
from textutils import read_text_file, process_text_files
content = read_text_file("document.txt")
results = process_text_files("./texts/", "report.txt")

## Тестирование пакета

Создадим простой тестовый скрипт для проверки работоспособности:
```
```

```python
# test_textutils.py
import sys
import os

# Добавляем путь к нашему пакету
sys.path.insert(0, os.path.dirname(os.path.abspath(__file__)))

from textutils import *

def test_analyzer():
    """Тестируем функции анализа"""
    text = "Привет мир! Это тестовый текст. Содержит три предложения?"

    print("=== Тестирование анализатора ===")
    print(f"Текст: {text}")
    print(f"Слов: {count_words(text)}")
    print(f"Символов: {count_characters(text)}")
    print(f"Предложений: {count_sentences(text)}")

    stats = text_statistics(text)
    print(f"Полная статистика: {stats}")
    print()

def test_cleaner():
    """Тестируем функции очистки"""
    dirty_text = "  Привет,    мир!!!   Как дела??  "

    print("=== Тестирование очистителя ===")
    print(f"Исходный текст: '{dirty_text}'")
    print(f"Без лишних пробелов: '{remove_extra_spaces(dirty_text)}'")
    print(f"Без пунктуации: '{remove_punctuation(dirty_text)}'")
    print(f"Полная очистка: '{clean_text(dirty_text)}'")
    print()

def test_file_operations():
    """Тестируем работу с файлами"""
    test_content = "Это тестовый файл.\nСодержит несколько строк.\nДля проверки работы пакета."

    print("=== Тестирование файловых операций ===")

    # Записываем тестовый файл
    if write_text_file("test.txt", test_content):
        print("Тестовый файл создан успешно")

    # Читаем файл обратно
    content = read_text_file("test.txt")
    if content:
        print(f"Файл прочитан, содержимое: {len(content)} символов")
```

```
        # Анализируем содержимое
        stats = text_statistics(content)
      print(f"Статистика файла: {stats['words']} слов, {stats['sentences']}
предложений")

    # Удаляем тестовый файл
    try:
        os.remove("test.txt")
        print("Тестовый файл удален")
    except:
        pass
    print()

if __name__ == "__main__":
    test_analyzer()
    test_cleaner()
    test_file_operations()
    print("Все тесты пройдены успешно!")
```

Подготовка к публикации

Для подготовки пакета к публикации на PyPI необходимо выполнить следующие шаги.

1. **Проверка структуры пакета**
   Убедитесь, что все файлы на месте и код работает корректно. Запустите тестовый скрипт для проверки функциональности.

2. **Создание дистрибутива**
   В корневой папке пакета выполните команду:

   ```
   python setup.py sdist bdist_wheel
   ```

   Эта команда создает папку `dist` с файлами для распространения.

3. **Проверка пакета**
   Используйте инструмент `twine` для проверки пакета перед публикацией:

   ```
   pip install twine
   twine check dist/*
   ```

4. **Тестовая публикация**
   Сначала опубликуйте пакет в тестовом репозитории PyPI:

   ```
   twine upload --repository-url https://test.pypi.org/legacy/
   dist/*
   ```

5. **Публикация в основном репозитории**
   После успешного тестирования опубликуйте в основном PyPI:

   ```
   twine upload dist/*
   ```

Заключение

В результате этой практической работы мы создали полноценный Python-пакет, который демонстрирует все основные принципы организации кода в модули и пакеты. Наш пакет `textutils` включает несколько модулей с четко разделенной функциональностью, имеет правильную структуру файлов, содержит документацию и готов к распространению.

Ключевые навыки, которые мы применили:

- **Модульная организация кода** — разделение функциональности по тематическим модулям.
- **Создание публичного API** — определение интерфейса пакета через `__init__.py`.
- **Документирование кода** — комментарии к функциям и общая документация пакета.
- **Структура пакета** — использование стандартных соглашений Python-сообщества.
- **Подготовка к публикации** — создание `setup.py` и дистрибутивов.

Этот опыт показывает, как небольшие утилитарные функции можно организовать в повторно используемый пакет, который принесет пользу не только вам, но и другим разработчикам. В реальной разработке создание собственных пакетов — это важный шаг к написанию профессионального кода и построению карьеры в программировании.

Глава 9. Объектно-ориентированное программирование

9.1. Философия ООП: моделирование реального мира

Представьте, что вы архитектор, проектирующий новый жилой комплекс. Вместо того чтобы рисовать каждый дом отдельно, вы создаете типовые проекты — шаблоны для разных видов жилья. Один шаблон для студии, другой для двухкомнатной квартиры, третий для таунхауса. Каждый конкретный дом строится по одному из этих шаблонов, но имеет свой уникальный адрес, своих жильцов и свою историю.

Объектно-ориентированное программирование (object-oriented programming, ООР) работает по тому же принципу. Это парадигма программирования, которая организует код вокруг объектов — сущностей, объединяющих данные и функции для работы с этими данными. Если до сих пор мы писали программы как последовательность инструкций, то теперь мы будем моделировать реальный мир через взаимодействие объектов.

Зачем нужно объектно-ориентированное программирование

В реальной жизни мы естественным образом думаем об окружающем мире в терминах объектов. Автомобиль — это не просто набор функций «ехать», «тормозить», «поворачивать», это целостная сущность со своими характеристиками: марка, цвет, объем двигателя, пробег. При этом каждый конкретный автомобиль уникален, хотя и создан по общему «шаблону» своей модели.

Процедурное программирование, которое мы изучали ранее, отлично подходит для решения вычислительных задач или создания простых утилит. Но когда программа становится сложной, когда нужно моделировать множество взаимосвязанных сущностей, процедурный подход начинает показывать свои ограничения:

```
# Процедурный подход - данные отделены от функций
user_name = "Анна Иванова"
user_email = "anna@example.com"
user_balance = 5000.0
def withdraw_money(name, email, balance, amount):
    if balance >= amount:
        return balance - amount
    else:
        print(f"Недостаточно средств у пользователя {name}")
        return balance
def deposit_money(name, email, balance, amount):
    return balance + amount
# Использование
user_balance = withdraw_money(user_name, user_email, user_balance, 1000)
```

Проблемы такого подхода становятся очевидными при масштабировании: данные передаются между функциями как отдельные параметры, легко перепутать по-

рядок аргументов, сложно поддерживать целостность данных, нет четкой связи между данными и операциями над ними.

Основные принципы ООП

Объектно-ориентированное программирование строится на четырех фундаментальных принципах, которые помогают создавать более организованный, понятный и поддерживаемый код.

Инкапсуляция (encapsulation) — это объединение данных и методов, работающих с этими данными, в единое целое. Кроме того, инкапсуляция подразумевает сокрытие внутренней реализации объекта от внешнего мира. Вы можете пользоваться телефоном, не зная, как именно работает его процессор или каким образом передаются сигналы по сотовой сети.

Наследование (inheritance) позволяет создавать новые классы на основе существующих, наследуя их свойства и поведение. Это как биологическая наследственность — дети наследуют черты родителей, но могут иметь и свои уникальные особенности.

Полиморфизм (polymorphism) означает способность объектов разных типов отвечать на одинаковые запросы каждый по-своему. Кнопка «Воспроизвести» работает одинаково на музыкальном плеере, видеоплеере и диктофоне, но каждое устройство выполняет это действие по-своему.

Абстракция (abstraction) — это выделение существенных характеристик объекта и игнорирование несущественных деталей. Карта города — это абстракция реального города, где показаны только важные для навигации элементы: улицы, здания, парки.

Моделирование реального мира

Рассмотрим, как объектно-ориентированный подход позволяет естественно моделировать реальные сущности. Представим систему управления библиотекой. В реальной библиотеке есть книги, читатели, библиотекари. У каждой сущности есть свои характеристики и свое поведение.

Книга имеет название, автора, ISBN, количество страниц. Книгу можно выдать читателю, вернуть в библиотеку, найти по автору или названию.

Читатель имеет имя, читательский билет, список взятых книг. Читатель может взять книгу, вернуть книгу, продлить срок аренды.

Библиотекарь может регистрировать новых читателей, добавлять книги в каталог, отслеживать просроченные возвраты.

В объектно-ориентированном программировании каждая такая сущность становится классом — шаблоном для создания конкретных объектов. Конкретная книга «Война и мир» Толстого или конкретный читатель Иван Петров становятся объектами — экземплярами соответствующих классов.

```
# Концептуальный пример (пока без реализации)
# Мы создаем объекты, которые естественно отражают реальный мир
book1 = Book("Война и мир", "Толстой Л.Н.", "978-5-17-123456-7")
reader1 = Reader("Иван Петров", "RD001234")
# Объекты взаимодействуют друг с другом
reader1.borrow_book(book1)
```

Преимущества объектно-ориентированного подхода

Модульность — программа разбивается на логические блоки (классы), каждый из которых отвечает за определенную функциональность. Это упрощает понимание, разработку и отладку кода.

Повторное использование кода — однажды созданный класс может использоваться в разных частях программы и даже в других проектах. Класс «Дата» пригодится в системе управления проектами, в календарном приложении, в системе учета заказов.

Легкость сопровождения — изменения в одном классе не влияют на другие части программы, если интерфейс класса остается неизменным. Вы можете улучшить алгоритм сортировки в классе «СписокКниг», и это не потребует изменений в классах «Читатель» или «Библиотекарь».

Масштабируемость — новую функциональность легко добавить, создав новые классы или расширив существующие через наследование. Нужна система заказа книг онлайн? Создайте класс «ОнлайнЧитатель», наследующий от «Читатель».

Соответствие ментальной модели — объектно-ориентированный код легче понимать, потому что он отражает то, как мы естественно думаем о мире в терминах объектов и их взаимодействий.

ООП не является универсальным решением всех проблем программирования. Для простых скриптов или математических вычислений процедурный подход может быть более уместным. Но когда программа моделирует сложную предметную область, когда нужно управлять состоянием множества взаимосвязанных сущностей, объектно-ориентированный подход показывает свою силу.

В следующем разделе мы изучим, как в Python создаются классы — шаблоны для объектов, и как из этих шаблонов рождаются конкретные экземпляры со своими уникальными характеристиками.

9.2. Классы и объекты: шаблоны и экземпляры

Класс в программировании — это как чертеж здания или рецепт блюда. Он описывает структуру и поведение, но сам по себе не является готовой постройкой или готовым обедом. Объект — это конкретная реализация класса, как конкретный дом, построенный по чертежу, или конкретный торт, испеченный по рецепту.

Анатомия класса в Python

В Python класс создается с помощью ключевого слова `class`. Давайте начнем с простейшего примера — класса для представления книги:

```
class Book:
    """Класс для представления книги"""
    pass  # Пока что пустой класс
```

Это минимальное определение класса. Ключевое слово `class` сообщает Python, что мы создаем новый класс. Book — имя класса (по соглашению имена классов пишутся в стиле CamelCase, начиная с заглавной буквы). Строка документации описывает назначение класса. `pass` — это «пустая» инструкция, которая ничего не делает, но синтаксически необходима для создания пустого блока кода.

Теперь мы можем создать объекты этого класса:

```
# Создание объектов (экземпляров) класса Book
book1 = Book()
book2 = Book()
print(type(book1))  # <class '__main__.Book'>
print(book1 == book2)  # False - это разные объекты
```

Каждый вызов `Book()` создает новый объект — уникальный экземпляр класса Book. Даже если они выглядят одинаково, это разные объекты в памяти компьютера.

Инициализация объектов: метод `__init__`

Пустой класс малополезен. Обычно при создании объекта мы хотим задать его начальные характеристики. Для этого используется специальный метод `__init__` (от initialize — инициализировать):

```
class Book:
    """Класс для представления книги"""

    def __init__(self, title, author, pages):
        """Инициализация объекта книги"""
        self.title = title        # Название книги
        self.author = author      # Автор книги
        self.pages = pages        # Количество страниц
        self.is_borrowed = False  # По умолчанию книга не выдана
```

Метод `__init__` — это конструктор класса, который автоматически вызывается при создании нового объекта. Обратите внимание на несколько важных моментов.

- Первый параметр `self` — это ссылка на создаваемый объект. Через `self` мы обращаемся к атрибутам и методам конкретного экземпляра.
- `self.title`, `self.author`, `self.pages` — это атрибуты экземпляра. Каждый объект класса Book будет иметь свои собственные значения этих атрибутов.
- `self.is_borrowed = False` показывает, что мы можем не только задавать значения атрибутов через параметры, но и устанавливать значения по умолчанию.

Теперь мы можем создавать книги с конкретными характеристиками:

```
# Создание объектов с начальными значениями
war_and_peace = Book("Война и мир", "Толстой Л.Н.", 1300)
python_guide = Book("Python Programming", "Smith J.", 450)
# Доступ к атрибутам объектов
print(f"Название: {war_and_peace.title}")       # Название: Война и мир
print(f"Автор: {war_and_peace.author}")         # Автор: Толстой Л.Н.
print(f"Страниц: {war_and_peace.pages}")        # Страниц: 1300
print(f"Выдана: {war_and_peace.is_borrowed}")   # Выдана: False
print(f"Название: {python_guide.title}")        # Название: Python Programming
```

Атрибуты экземпляра

Атрибуты экземпляра — это переменные, которые принадлежат конкретному объекту. Каждый объект имеет свою собственную копию этих атрибутов:

```
# Создаем два объекта
book1 = Book("Мастер и Маргарита", "Булгаков М.А.", 480)
book2 = Book("1984", "Оруэлл Дж.", 320)
# У каждого объекта свои значения атрибутов
print(book1.title)  # Мастер и Маргарита
print(book2.title)  # 1984
# Изменяем атрибут одного объекта
book1.is_borrowed = True
print(book1.is_borrowed)  # True
print(book2.is_borrowed)  # False - изменение не затронуло другой объект
```

Мы можем не только читать значения атрибутов, но и изменять их, а также добавлять новые атрибуты динамически:

```
book1.rating = 5  # Добавляем новый атрибут
book1.notes = "Отличная книга!"  # Еще один атрибут
print(book1.rating)  # 5
# print(book2.rating)  # AttributeError - у book2 нет атрибута rating
```

Методы класса

Методы — это функции, которые принадлежат классу и определяют поведение объектов. Добавим к нашему классу Book несколько полезных методов:

```
class Book:
    """Класс для представления книги"""

    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
        self.is_borrowed = False

    def borrow(self):
        """Выдать книгу читателю"""
        if self.is_borrowed:
            print(f"Книга '{self.title}' уже выдана")
        else:
            self.is_borrowed = True
            print(f"Книга '{self.title}' выдана читателю")

    def return_book(self):
        """Вернуть книгу в библиотеку"""
        if self.is_borrowed:
            self.is_borrowed = False
            print(f"Книга '{self.title}' возвращена в библиотеку")
        else:
            print(f"Книга '{self.title}' не была выдана")

    def get_info(self):
        """Получить информацию о книге"""
        status = "выдана" if self.is_borrowed else "доступна"
       return f"'{self.title}' - {self.author}, {self.pages} стр., {status}"
```

Каждый метод класса должен принимать `self` как первый параметр — это соглашение Python. Через `self` метод получает доступ к атрибутам и другим методам того объекта, для которого он вызывается:

```
# Создаем объект и используем его методы
my_book = Book("Преступление и наказание", "Достоевский Ф.М.", 671)
print(my_book.get_info())
# 'Преступление и наказание' - Достоевский Ф.М., 671 стр., доступна
my_book.borrow()           # Книга 'Преступление и наказание' выдана читателю
print(my_book.get_info())
# 'Преступление и наказание' - Достоевский Ф.М., 671 стр., выдана
my_book.return_book()      # Книга 'Преступление и наказание' возвращена в биб-
лиотеку
```

Создание более сложного класса

Рассмотрим более сложный пример — класс для представления читателя библиотеки:

```
class Reader:
    """Класс для представления читателя библиотеки"""

    def __init__(self, name, reader_id):
        self.name = name
        self.reader_id = reader_id
        self.borrowed_books = []  # Список взятых книг
        self.max_books = 5        # Максимальное количество книг

    def borrow_book(self, book):
        """Взять книгу из библиотеки"""
        if len(self.borrowed_books) >= self.max_books:
           print(f"{self.name} не может взять больше {self.max_books} книг")
            return

        if book.is_borrowed:
            print(f"Книга '{book.title}' уже выдана другому читателю")
            return

        book.borrow()
        self.borrowed_books.append(book)
        print(f"{self.name} взял книгу '{book.title}'")

    def return_book(self, book):
        """Вернуть книгу в библиотеку"""
        if book in self.borrowed_books:
            book.return_book()
            self.borrowed_books.remove(book)
            print(f"{self.name} вернул книгу '{book.title}'")
        else:
            print(f"{self.name} не брал книгу '{book.title}'")
```

```
    def list_books(self):
        """Показать список взятых книг"""
        if not self.borrowed_books:
            print(f"У {self.name} нет взятых книг")
        else:
            print(f"Книги у {self.name}:")
            for book in self.borrowed_books:
                print(f"  - {book.get_info()}")
```

Теперь мы можем создать читателя и смоделировать процесс выдачи и возврата книг:

```
# Создаем читателя и книги
reader = Reader("Анна Иванова", "RD12345")
book1 = Book("Гарри Поттер", "Роулинг Дж.", 400)
book2 = Book("Властелин колец", "Толкин Дж.", 1200)
# Читатель берет книги
reader.borrow_book(book1)  # Анна Иванова взял книгу 'Гарри Поттер'
reader.borrow_book(book2)  # Анна Иванова взял книгу 'Властелин колец'
# Проверяем список взятых книг
reader.list_books()
# Книги у Анна Иванова:
#   - 'Гарри Поттер' - Роулинг Дж., 400 стр., выдана
#   - 'Властелин колец' - Толкин Дж., 1200 стр., выдана
# Возвращаем одну книгу
reader.return_book(book1)  # Анна Иванова вернул книгу 'Гарри Поттер'
```

Важные концепции

Инкапсуляция данных: каждый объект содержит свои данные (атрибуты) и функции для работы с ними (методы). Внешний код взаимодействует с объектом через его публичный интерфейс — методы класса.

Состояние объекта: совокупность значений всех атрибутов объекта в конкретный момент времени. Методы объекта могут изменять его состояние.

Идентичность объекта: каждый объект уникален, даже если его атрибуты совпадают с атрибутами другого объекта:

```
book1 = Book("1984", "Оруэлл Дж.", 300)
book2 = Book("1984", "Оруэлл Дж.", 300)
print(book1 == book2)      # False - разные объекты
print(book1 is book2)      # False - разные объекты в памяти
print(id(book1) == id(book2))  # False - разные идентификаторы
```

Жизненный цикл объекта: объект создается при вызове класса, существует в памяти, пока на него есть ссылки, и автоматически удаляется сборщиком мусора, когда становится недоступным.

В следующих разделах мы изучим, как правильно организовать атрибуты и методы в классах, как скрывать внутреннюю реализацию от внешнего мира и как создавать иерархии классов через наследование. Эти концепции превратят наши простые классы в мощный инструмент для моделирования сложных систем реального мира.

9.3. Атрибуты и методы: состояние и поведение

В предыдущих разделах мы познакомились с классами как шаблонами для создания объектов. Теперь пришло время глубже изучить внутреннюю структуру объектов — их **атрибуты** и **методы**. Если объект — это модель реального предмета или абстрактной сущности, то атрибуты описывают его **состояние (`state`)**, а методы определяют его **поведение (`behavior`)**.

Представьте автомобиль: его состояние характеризуется такими параметрами как текущая скорость, уровень топлива, включенность фар, а поведение включает действия вроде ускорения, торможения, поворота. В программировании эта аналогия работает точно так же — атрибуты хранят данные, а методы содержат код, который может эти данные обрабатывать.

Методы экземпляра: поведение объекта

Методы экземпляра (instance methods) — это функции, определенные внутри класса, которые работают с конкретным экземпляром объекта. Они имеют доступ к атрибутам этого экземпляра и могут их изменять. Каждый метод экземпляра обязательно принимает первым параметром ссылку на сам объект, которая по соглашению называется `self`.

```
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self.balance = initial_balance
        self.transaction_history = []

    def deposit(self, amount):
        """Метод для внесения средств на счет"""
        if amount > 0:
            self.balance += amount
            self.transaction_history.append(f"Deposit: +{amount}")
            print(f"Deposited {amount}. New balance: {self.balance}")
        else:
            print("Deposit amount must be positive")

    def withdraw(self, amount):
        """Метод для снятия средств со счета"""
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            self.transaction_history.append(f"Withdrawal: -{amount}")
            print(f"Withdrew {amount}. New balance: {self.balance}")
        elif amount > self.balance:
            print("Insufficient funds")
        else:
            print("Withdrawal amount must be positive")

    def get_balance(self):
        """Метод для получения текущего баланса"""
        return self.balance
```

```
    def get_statement(self):
        """Метод для получения выписки по счету"""
        print(f"Account: {self.account_number}")
        print(f"Current balance: {self.balance}")
        print("Transaction history:")
        for transaction in self.transaction_history:
            print(f"  {transaction}")
```

В этом примере каждый метод работает с конкретным экземпляром банковского счета. Параметр `self` позволяет методу обращаться к атрибутам именно того объекта, для которого он был вызван:

```
# Создаем два разных счета
account1 = BankAccount("123-456-789", 1000)
account2 = BankAccount("987-654-321", 500)
# Операции с первым счетом не влияют на второй
account1.deposit(200)  # Deposited 200. New balance: 1200
account2.withdraw(100) # Withdrew 100. New balance: 400
print(account1.get_balance())  # 1200
print(account2.get_balance())  # 400
```

Загадочный `self`: понимание механизма

Параметр `self` часто вызывает недоумение у начинающих программистов. Важно понимать, что `self` — это не ключевое слово Python, а просто соглашение об именовании. Технически вы можете назвать этот параметр как угодно, но использование `self` является стандартом, которого придерживается всё сообщество Python.

Когда вы вызываете метод объекта, Python автоматически передает ссылку на этот объект в качестве первого аргумента:

```
# Эти два вызова эквивалентны:
account1.deposit(100)
BankAccount.deposit(account1, 100)
```

Первая форма — это синтаксический сахар, который делает код более читаемым. Вторая форма показывает, что на самом деле происходит под капотом: метод класса вызывается с явной передачей экземпляра.

Атрибуты экземпляра vs атрибуты класса

Существует важное различие между **атрибутами экземпляра (instance attributes)** и **атрибутами класса (class attributes)**. Атрибуты экземпляра принадлежат конкретному объекту и могут различаться у разных экземпляров. Атрибуты класса принадлежат всему классу и являются общими для всех его экземпляров.

```
class Student:
    # Атрибут класса - общий для всех студентов
    university_name = "Python University"
    student_count = 0
```

```
    def __init__(self, name, student_id):
        # Атрибуты экземпляра - уникальные для каждого студента
        self.name = name
        self.student_id = student_id
        self.grades = []

        # Увеличиваем счетчик студентов
        Student.student_count += 1

    def add_grade(self, grade):
        """Добавить оценку студенту"""
        if 0 <= grade <= 100:
            self.grades.append(grade)
        else:
            print("Grade must be between 0 and 100")

    def get_average_grade(self):
        """Вычислить средний балл"""
        if self.grades:
            return sum(self.grades) / len(self.grades)
        return 0

    def get_info(self):
        """Получить информацию о студенте"""
        avg_grade = self.get_average_grade()
          return f"Student: {self.name} (ID: {self.student_id}), Average:
{avg_grade:.2f}"

    @classmethod
    def get_student_count(cls):
        """Метод класса для получения общего количества студентов"""
        return cls.student_count
```

Обратите внимание на различия в использовании:

```
# Создаем студентов
alice = Student("Alice Johnson", "ST001")
bob = Student("Bob Smith", "ST002")
# Атрибуты экземпляра уникальны
alice.add_grade(95)
alice.add_grade(87)
bob.add_grade(78)
bob.add_grade(92)
print(alice.get_info())   # Student: Alice Johnson (ID: ST001), Average:
91.00
print(bob.get_info())    # Student: Bob Smith (ID: ST002), Average: 85.00
# Атрибуты класса общие для всех
print(alice.university_name)    # Python University
print(bob.university_name)      # Python University
print(Student.student_count)    # 2
```

Изменение атрибутов класса

Важно понимать разницу между изменением атрибута класса и созданием атрибута экземпляра с тем же именем:

```
class Counter:
    value = 0  # Атрибут класса

    def increment_class(self):
        Counter.value += 1  # Изменяем атрибут класса

    def increment_instance(self):
        self.value += 1     # Создаем/изменяем атрибут экземпляра
counter1 = Counter()
counter2 = Counter()
print(counter1.value)  # 0 (атрибут класса)
print(counter2.value)  # 0 (атрибут класса)
counter1.increment_class()
print(counter1.value)  # 1 (атрибут класса изменился для всех)
print(counter2.value)  # 1 (атрибут класса изменился для всех)
counter1.increment_instance()
print(counter1.value)  # 2 (теперь у counter1 есть собственный атрибут)
print(counter2.value)  # 1 (у counter2 все еще атрибут класса)
counter2.increment_class()
print(counter1.value)  # 2 (собственный атрибут counter1 не изменился)
print(counter2.value)  # 2 (атрибут класса увеличился)
```

Методы как атрибуты объекта

В Python методы — это тоже атрибуты объекта, только специального типа. Их можно присваивать переменным и передавать в другие функции:

```
class Calculator:
    def __init__(self, initial_value=0):
        self.value = initial_value

    def add(self, x):
        self.value += x
        return self.value

    def multiply(self, x):
        self.value *= x
        return self.value
calc = Calculator(10)
# Методы можно присваивать переменным
add_operation = calc.add
multiply_operation = calc.multiply
# И вызывать через эти переменные
print(add_operation(5))      # 15
print(multiply_operation(3)) # 45
```

Динамическое добавление атрибутов и методов

Python позволяет динамически добавлять атрибуты и методы к объектам во время выполнения программы:

```
class FlexibleClass:
    def __init__(self, name):
        self.name = name
obj = FlexibleClass("test")
# Добавляем новый атрибут
obj.new_attribute = "Dynamic value"
print(obj.new_attribute)  # Dynamic value
# Добавляем новый метод (осторожно с этим подходом!)
def dynamic_method(self):
    return f"Hello from {self.name}"
obj.say_hello = dynamic_method.__get__(obj, FlexibleClass)
print(obj.say_hello())  # Hello from test
```

Хотя такая гибкость может показаться заманчивой, в реальном программировании ее следует использовать с осторожностью, так как она может сделать код непредсказуемым и трудным для понимания.

9.4. Инкапсуляция: сокрытие внутренней реализации

Инкапсуляция (encapsulation) — один из фундаментальных принципов объектно-ориентированного программирования, который заключается в сокрытии внутренних деталей реализации объекта от внешнего мира. Этот принцип помогает создавать надежные и поддерживаемые программы, защищая внутреннее состояние объекта от некорректных изменений.

Представьте телевизор: вы можете изменять громкость, переключать каналы, включать и выключать его с помощью пульта, но вы не можете (и не должны) напрямую изменять напряжение на процессоре или частоту сигнала. Внутренние компоненты скрыты, а доступ к функциональности предоставляется через четко определенный интерфейс.

Проблема прямого доступа к атрибутам

Рассмотрим пример без использования инкапсуляции:

```
class BankAccountInsecure:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self.balance = initial_balance
# Создаем счет
account = BankAccountInsecure("123-456", 1000)
# Прямой доступ к атрибутам может привести к проблемам
account.balance = -500    # Отрицательный баланс!
account.balance = "много" # Некорректный тип данных!
print(account.balance)    # много
```

Такой подход создает множество проблем: нет контроля корректности данных, отсутствует логика валидации, невозможно отследить изменения, нарушается целостность объекта.

Приватные атрибуты: соглашения Python

В отличие от многих языков программирования, Python не имеет встроенного механизма для создания действительно приватных атрибутов. Вместо этого используются **соглашения об именовании (naming conventions)**, которые сигнализируют о предназначении атрибута.

Одинарное подчеркивание: «защищенные» атрибуты

Атрибуты и методы, имена которых начинаются с одного подчеркивания, считаются **защищенными (protected)**. Это соглашение означает: «Я предназначен для внутреннего использования, но при необходимости к нему можно обращаться»:

```
class BankAccountProtected:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self._balance = initial_balance  # "Защищенный" атрибут
        self._transaction_count = 0

    def deposit(self, amount):
        if self._validate_amount(amount):
            self._balance += amount
            self._transaction_count += 1
            self._log_transaction(f"Deposit: +{amount}")

    def withdraw(self, amount):
        if self._validate_amount(amount) and amount <= self._balance:
            self._balance -= amount
            self._transaction_count += 1
            self._log_transaction(f"Withdrawal: -{amount}")
        else:
            print("Invalid withdrawal")

    def _validate_amount(self, amount):
        """Защищенный метод для валидации суммы"""
        return isinstance(amount, (int, float)) and amount > 0

    def _log_transaction(self, description):
        """Защищенный метод для логирования"""
        print(f"Transaction #{self._transaction_count}: {description}")

    def get_balance(self):
        return self._balance
```

Двойное подчеркивание: «приватные» атрибуты

Атрибуты, начинающиеся с двойного подчеркивания, подвергаются **искажению имени (name mangling)**. Python автоматически изменяет их имена, делая прямой доступ более сложным:

```
class SecureBankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self.__balance = initial_balance      # "Приватный" атрибут
        self.__secret_key = "ABC123"         # Реально приватная информация

    def deposit(self, amount):
        if self.__validate_transaction(amount):
            self.__balance += amount
            print(f"Deposited {amount}. New balance: {self.__balance}")

    def withdraw(self, amount):
       if self.__validate_transaction(amount) and amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount}. New balance: {self.__balance}")
        else:
            print("Transaction denied")

    def __validate_transaction(self, amount):
        """Приватный метод валидации"""
        return isinstance(amount, (int, float)) and amount > 0

    def get_balance(self):
        return self.__balance
account = SecureBankAccount("123-456", 1000)
# Обычный доступ не работает
# print(account.__balance)  # AttributeError
# Но искаженное имя все же доступно (хотя не рекомендуется)
print(account._SecureBankAccount__balance)  # 1000
```

Свойства: элегантный контроль доступа

Свойства (properties) предоставляют элегантный способ контроля доступа к атрибутам, позволяя использовать синтаксис обращения к атрибуту, но при этом выполнять дополнительную логику:

```
class SmartBankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self._balance = 0
        self.balance = initial_balance  # Используем property для установки

    @property
    def balance(self):
        """Геттер для баланса"""
        return self._balance

    @balance.setter
    def balance(self, value):
        """Сеттер для баланса с валидацией"""
        if not isinstance(value, (int, float)):
            raise TypeError("Balance must be a number")
```

```
        if value < 0:
            raise ValueError("Balance cannot be negative")
        self._balance = value

    @property
    def balance_status(self):
        """Вычисляемое свойство только для чтения"""
        if self._balance >= 10000:
            return "Premium"
        elif self._balance >= 1000:
            return "Standard"
        else:
            return "Basic"
# Использование выглядит как работа с обычным атрибутом
account = SmartBankAccount("123-456", 1500)
print(account.balance)        # 1500
print(account.balance_status) # Standard
account.balance = 2000        # Вызывается setter
print(account.balance)        # 2000
# Но валидация работает
try:
    account.balance = -100    # ValueError: Balance cannot be negative
except ValueError as e:
    print(f"Error: {e}")
```

Геттеры и сеттеры: классический подход

Иногда более явный подход с методами-геттерами и сеттерами может быть предпочтительным, особенно когда требуется сложная логика:

```
class AdvancedBankAccount:
    def __init__(self, account_number, initial_balance=0):
        self._account_number = account_number
        self._balance = 0
        self._is_frozen = False
        self._daily_withdrawal_limit = 5000
        self._daily_withdrawn = 0
        self.set_balance(initial_balance)

    def get_balance(self):
        """Получить текущий баланс"""
        if self._is_frozen:
            print("Account is frozen. Contact support.")
            return None
        return self._balance

    def set_balance(self, amount):
        """Установить баланс с проверками"""
        if not isinstance(amount, (int, float)):
            raise TypeError("Amount must be numeric")
        if amount < 0:
```

```
            raise ValueError("Balance cannot be negative")
        self._balance = amount

    def withdraw(self, amount):
        """Снять деньги с учетом дневного лимита"""
        if self._is_frozen:
            print("Account is frozen")
            return False

        if self._daily_withdrawn + amount > self._daily_withdrawal_limit:
             remaining_limit = self._daily_withdrawal_limit - self._daily_
withdrawn
          print(f"Daily limit exceeded. Remaining limit: {remaining_limit}")
            return False

        if amount > self._balance:
            print("Insufficient funds")
            return False

        self._balance -= amount
        self._daily_withdrawn += amount
        print(f"Withdrew {amount}. Remaining balance: {self._balance}")
        return True

    def freeze_account(self):
        """Заморозить счет"""
        self._is_frozen = True
        print("Account has been frozen")

    def unfreeze_account(self):
        """Разморозить счет"""
        self._is_frozen = False
        print("Account has been unfrozen")

    def reset_daily_limit(self):
        """Сбросить дневной лимит (обычно вызывается в полночь)"""
        self._daily_withdrawn = 0
```

9.5. Наследование: построение иерархий классов

Наследование (inheritance) — это механизм объектно-ориентированного программирования, который позволяет создавать новые классы на основе уже существующих. Дочерний класс автоматически получает все атрибуты и методы родительского класса, при этом может добавлять свои собственные или изменять унаследованные.

Наследование отражает естественные иерархические отношения, которые мы наблюдаем в реальном мире. Например, «автомобиль» — это частный случай «транспортного средства», а «спортивный автомобиль» — частный случай «автомобиля». Каждый уровень иерархии добавляет новые характеристики, сохраняя при этом все свойства вышестоящих уровней.

Базовые и дочерние классы

Рассмотрим классический пример с животными, который демонстрирует принцип наследования:

```
class Animal:
    """Базовый класс для всех животных"""

    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.energy = 100

    def eat(self, food_amount):
        """Базовый метод питания"""
        self.energy += food_amount
        print(f"{self.name} ate and gained {food_amount} energy")

    def sleep(self):
        """Базовый метод сна"""
        self.energy = 100
        print(f"{self.name} slept and restored energy")

    def make_sound(self):
        """Базовый метод издавания звука"""
        print(f"{self.name} makes a generic animal sound")

    def get_info(self):
        """Получить информацию о животном"""
        return f"{self.name} is a {self.species} with {self.energy} energy"
# Создаем дочерний класс
class Dog(Animal):
    """Класс собаки, наследующий от Animal"""

    def __init__(self, name, breed):
        # Вызываем конструктор родительского класса
        super().__init__(name, "Dog")
        self.breed = breed
        self.tricks = []

    def make_sound(self):
        """Переопределяем метод издавания звука"""
        print(f"{self.name} barks: Woof! Woof!")

    def learn_trick(self, trick):
        """Новый метод, специфичный для собак"""
        self.tricks.append(trick)
        print(f"{self.name} learned a new trick: {trick}")

    def perform_tricks(self):
        """Выполнить все изученные трюки"""
        if self.tricks:
```

```
            print(f"{self.name} performs: {', '.join(self.tricks)}")
        else:
            print(f"{self.name} doesn't know any tricks yet")
```

В этом примере класс Dog **наследует** от класса Animal. Это означает, что собака автоматически получает все методы и атрибуты животного, но может добавлять свои специфические особенности:

```
# Создаем экземпляр собаки
buddy = Dog("Buddy", "Golden Retriever")
# Используем унаследованные методы
buddy.eat(20)          # Buddy ate and gained 20 energy
print(buddy.get_info()) # Buddy is a Dog with 120 energy
# Используем переопределенный метод
buddy.make_sound()     # Buddy barks: Woof! Woof!
# Используем специфичные для собаки методы
buddy.learn_trick("sit")
buddy.learn_trick("fetch")
buddy.perform_tricks() # Buddy performs: sit, fetch
```

Функция super(): связь с родительским классом

Функция super() предоставляет доступ к методам родительского класса. Это особенно важно при переопределении методов, когда нужно расширить, а не полностью заменить родительское поведение:

```
class Cat(Animal):
    """Класс кота, демонстрирующий использование super()"""

    def __init__(self, name, indoor=True):
        super().__init__(name, "Cat")
        self.indoor = indoor
        self.lives = 9

    def make_sound(self):
        """Переопределяем звук"""
        print(f"{self.name} meows: Meow!")

    def eat(self, food_amount):
        """Расширяем родительский метод питания"""
        # Сначала выполняем базовое поведение
        super().eat(food_amount)

        # Добавляем специфичное для кота поведение
        if food_amount > 50:
            print(f"{self.name} is very satisfied and starts purring")

    def hunt(self):
        """Специфичный для кота метод"""
        if not self.indoor:
            self.energy -= 30
            print(f"{self.name} went hunting and spent 30 energy")
```

```
        else:
            print(f"{self.name} is an indoor cat and cannot hunt")
# Демонстрация работы
whiskers = Cat("Whiskers", indoor=False)
whiskers.eat(60)  # Выполнится и базовый метод, и дополнительное поведение
# Whiskers ate and gained 60 energy
# Whiskers is very satisfied and starts purring
whiskers.hunt()   # Whiskers went hunting and spent 30 energy
```

Построение многоуровневой иерархии

Наследование может быть многоуровневым, создавая глубокие иерархии классов:

```
class Vehicle:
    """Базовый класс транспортных средств"""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.fuel_level = 0

    def start_engine(self):
        print(f"{self.make} {self.model} engine started")

    def stop_engine(self):
        print(f"{self.make} {self.model} engine stopped")

    def refuel(self, amount):
        self.fuel_level += amount
        print(f"Refueled {amount} liters. Total: {self.fuel_level}")
class Car(Vehicle):
    """Класс автомобиля"""

    def __init__(self, make, model, year, doors):
        super().__init__(make, model, year)
        self.doors = doors
        self.passengers = 0

    def load_passengers(self, count):
        max_passengers = (self.doors - 2) * 2 + 2  # Примерная формула
        if count <= max_passengers:
            self.passengers = count
            print(f"Loaded {count} passengers")
        else:
         print(f"Cannot load {count} passengers. Maximum: {max_passengers}")

    def drive(self, distance):
        fuel_needed = distance * 0.08  # 8 литров на 100 км
        if self.fuel_level >= fuel_needed:
```

```
            self.fuel_level -= fuel_needed
                print(f"Drove {distance} km. Fuel remaining: {self.fuel_
level:.1f}")
        else:
            print("Not enough fuel for this trip")
class SportsCar(Car):
    """Класс спортивного автомобиля"""

    def __init__(self, make, model, year, max_speed):
       super().__init__(make, model, year, 2)  # У спорткаров обычно две двери
        self.max_speed = max_speed
        self.turbo_mode = False

    def activate_turbo(self):
        """Специфичный для спорткара метод"""
        self.turbo_mode = True
        print(f"{self.make} {self.model} turbo activated! Max speed: {self.
max_speed}")

    def drive(self, distance):
        """Переопределяем метод для учета турборежима"""
        fuel_multiplier = 1.5 if self.turbo_mode else 1.0
        fuel_needed = distance * 0.08 * fuel_multiplier

        if self.fuel_level >= fuel_needed:
            self.fuel_level -= fuel_needed
            speed_info = " (TURBO MODE)" if self.turbo_mode else ""
            print(f"Drove {distance} km{speed_info}. Fuel remaining: {self.
fuel_level:.1f}")
        else:
            print("Not enough fuel for this trip")
# Демонстрация трехуровневой иерархии
ferrari = SportsCar("Ferrari", "F8 Tributo", 2021, 340)
ferrari.refuel(100)           # Метод из Vehicle
ferrari.load_passengers(2)    # Метод из Car
ferrari.activate_turbo()      # Метод из SportsCar
ferrari.drive(50)             # Переопределенный метод
```

Множественное наследование

Python поддерживает **множественное наследование (multiple inheritance)**, когда класс может наследовать от нескольких родительских классов одновременно. Это мощный, но сложный механизм, требующий осторожного использования:

```
class Flyable:
    """Миксин для летающих объектов"""

    def __init__(self):
        self.altitude = 0
        self.flying = False
```

```
    def take_off(self):
        if not self.flying:
            self.flying = True
            self.altitude = 100
            print("Taking off... Now flying at 100m")
        else:
            print("Already flying")

    def land(self):
        if self.flying:
            self.flying = False
            self.altitude = 0
            print("Landing... Safely on the ground")
        else:
            print("Already on the ground")

    def change_altitude(self, new_altitude):
        if self.flying:
            self.altitude = new_altitude
            print(f"Changed altitude to {new_altitude}m")
        else:
            print("Cannot change altitude while on the ground")
class Swimmable:
    """Миксин для плавающих объектов"""

    def __init__(self):
        self.depth = 0
        self.swimming = False

    def dive(self, target_depth):
        self.swimming = True
        self.depth = target_depth
        print(f"Diving to {target_depth}m depth")

    def surface(self):
        if self.swimming:
            self.depth = 0
            self.swimming = False
            print("Surfacing... Now at water surface")
        else:
            print("Not in water")
class Duck(Animal, Flyable, Swimmable):
    """Утка - может летать и плавать"""

    def __init__(self, name):
        # Вызываем конструкторы всех родительских классов
        Animal.__init__(self, name, "Duck")
        Flyable.__init__(self)
        Swimmable.__init__(self)
        self.feathers_waterproof = True
```

```
    def make_sound(self):
        print(f"{self.name} quacks: Quack! Quack!")

    def migrate(self):
        """Специфичное поведение утки"""
        print(f"{self.name} is migrating...")
        self.take_off()
        self.change_altitude(1000)
        print("Flying to warmer regions")
# Использование множественного наследования
donald = Duck("Donald")
# Методы из Animal
donald.eat(30)              # Donald ate and gained 30 energy
donald.make_sound()         # Donald quacks: Quack! Quack!
# Методы из Flyable
donald.take_off()           # Taking off... Now flying at 100m
donald.change_altitude(500) # Changed altitude to 500m
# Методы из Swimmable
donald.land()               # Landing... Safely on the ground
donald.dive(5)              # Diving to 5m depth
donald.surface()            # Surfacing... Now at water surface
# Собственный метод
donald.migrate()            # Демонстрация комплексного поведения
```

Порядок разрешения методов (MRO)

При множественном наследовании важно понимать **порядок разрешения методов (Method Resolution Order, MRO)**. Python использует алгоритм C3-линеаризации для определения порядка поиска методов:

```
class A:
    def method(self):
        print("Method from A")
class B(A):
    def method(self):
        print("Method from B")
        super().method()
class C(A):
    def method(self):
        print("Method from C")
        super().method()
class D(B, C):
    def method(self):
        print("Method from D")
        super().method()
# Проверяем порядок разрешения методов
print(D.__mro__)  # Показывает порядок поиска методов
d = D()
d.method()
# Выведет:
# Method from D
```

```
# Method from B
# Method from C
# Method from A
```

9.6. Полиморфизм: одинаковый интерфейс, разное поведение

Полиморфизм (polymorphism) — это способность объектов разных классов реагировать на одинаковые методы по-разному. Буквально термин означает «множество форм» и описывает ситуацию, когда один и тот же интерфейс может иметь различные реализации в зависимости от типа объекта.

Представьте пульт от телевизора: кнопка «включить» работает одинаково для всех устройств, но каждое устройство включается по-своему — телевизор показывает изображение, музыкальный центр начинает воспроизводить звук, кондиционер начинает охлаждать воздух. Интерфейс одинаковый, но поведение различается.

Полиморфизм через наследование

Наиболее очевидный способ достижения полиморфизма — через наследование и переопределение методов:

```
class Shape:
    """Базовый класс геометрической фигуры"""

    def __init__(self, name):
        self.name = name

    def area(self):
        """Базовый метод расчета площади"""
        raise NotImplementedError("Subclass must implement area method")

    def perimeter(self):
        """Базовый метод расчета периметра"""
       raise NotImplementedError("Subclass must implement perimeter method")

    def describe(self):
        """Описание фигуры"""
         return f"{self.name}: area = {self.area():.2f}, perimeter = {self.
perimeter():.2f}"
class Rectangle(Shape):
    """Прямоугольник"""

    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

```
    def perimeter(self):
        return 2 * (self.width + self.height)
class Circle(Shape):
    """Окружность"""

    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius
class Triangle(Shape):
    """Треугольник"""

    def __init__(self, side_a, side_b, side_c):
        super().__init__("Triangle")
        self.side_a = side_a
        self.side_b = side_b
        self.side_c = side_c

    def area(self):
        # Формула Герона
        s = (self.side_a + self.side_b + self.side_c) / 2
          return (s * (s - self.side_a) * (s - self.side_b) * (s - self.
side_c)) ** 0.5

    def perimeter(self):
        return self.side_a + self.side_b + self.side_c
# Полиморфизм в действии
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Triangle(3, 4, 5)
]
# Один и тот же код работает с разными типами объектов
for shape in shapes:
    print(shape.describe())

# Выведет:
# Rectangle: area = 15.00, perimeter = 16.00
# Circle: area = 50.27, perimeter = 25.13
# Triangle: area = 6.00, perimeter = 12.00
```

Duck Typing: «Если это ходит как утка...»

В Python активно используется концепция **Duck Typing** — «если это ходит как утка и крякает как утка, то это утка». Это означает, что тип объекта менее важен, чем наличие у него нужных методов:

```
class FileWriter:
    """Класс для записи в файл"""

    def __init__(self, filename):
        self.filename = filename
        self.content = []

    def write(self, text):
        self.content.append(text)
        print(f"Writing to file {self.filename}: {text}")

    def close(self):
            print(f"File {self.filename} closed. Total lines: {len(self.
content)}")
class DatabaseLogger:
    """Класс для записи в базу данных"""

    def __init__(self, connection_string):
        self.connection_string = connection_string
        self.entries = []

    def write(self, text):
        self.entries.append(text)
        print(f"Logging to database: {text}")

    def close(self):
            print(f"Database connection closed. Total entries: {len(self.
entries)}")
class ConsoleOutput:
    """Класс для вывода в консоль"""

    def write(self, text):
        print(f"Console: {text}")

    def close(self):
        print("Console output finished")
def log_messages(writer, messages):
    """Функция, работающая с любым объектом, имеющим методы write и close"""
    for message in messages:
        writer.write(message)
    writer.close()
# Duck Typing в действии - разные типы, одинаковый интерфейс
file_writer = FileWriter("log.txt")
db_logger = DatabaseLogger("postgresql://localhost:5432/logs")
console = ConsoleOutput()
messages = ["Starting application", "Processing data", "Operation completed"]
# Одна и та же функция работает с разными типами объектов
log_messages(file_writer, messages)
print("-" * 30)
log_messages(db_logger, messages)
print("-" * 30)
log_messages(console, messages)
```

Переопределение методов для специфического поведения

Полиморфизм особенно мощен при создании семейств классов с общим интерфейсом, но различной реализацией:

```
class PaymentProcessor:
    """Базовый класс для обработки платежей"""

    def __init__(self, amount):
        self.amount = amount
        self.transaction_fee = 0

    def calculate_fee(self):
        """Базовый расчет комиссии"""
        return self.amount * 0.01  # 1% комиссия

    def process_payment(self):
        """Базовый метод обработки платежа"""
        self.transaction_fee = self.calculate_fee()
        print(f"Processing payment of ${self.amount:.2f}")
        print(f"Transaction fee: ${self.transaction_fee:.2f}")
        return True

    def send_confirmation(self):
        """Отправка подтверждения"""
        print("Payment confirmation sent")
class CreditCardProcessor(PaymentProcessor):
    """Обработка платежей по кредитным картам"""

    def __init__(self, amount, card_number, cvv):
        super().__init__(amount)
        self.card_number = card_number
        self.cvv = cvv

    def calculate_fee(self):
        """Кредитные карты имеют фиксированную комиссию"""
        return 2.50 if self.amount < 100 else self.amount * 0.025

    def process_payment(self):
        """Специфичная обработка для кредитных карт"""
        print(f"Processing credit card payment...")
        print(f"Card ending in {self.card_number[-4:]}")

        # Проверка CVV (упрощенная)
        if len(str(self.cvv)) != 3:
            print("Invalid CVV")
            return False

        return super().process_payment()
class PayPalProcessor(PaymentProcessor):
    """Обработка платежей через PayPal"""
```

```
    def __init__(self, amount, email):
        super().__init__(amount)
        self.email = email

    def calculate_fee(self):
        """PayPal имеет прогрессивную комиссию"""
        if self.amount <= 10:
            return 0.30 + self.amount * 0.029
        else:
            return self.amount * 0.034

    def process_payment(self):
        """Специфичная обработка для PayPal"""
        print(f"Processing PayPal payment for {self.email}")
        return super().process_payment()

    def send_confirmation(self):
        """PayPal отправляет подтверждение по email"""
        print(f"PayPal confirmation sent to {self.email}")
class CryptocurrencyProcessor(PaymentProcessor):
    """Обработка криптовалютных платежей"""

    def __init__(self, amount, wallet_address, currency="BTC"):
        super().__init__(amount)
        self.wallet_address = wallet_address
        self.currency = currency

    def calculate_fee(self):
        """Криптовалюты имеют низкую комиссию"""
        return max(0.001 * self.amount, 0.50)  # Минимум $0.50

    def process_payment(self):
        """Специфичная обработка для криптовалют"""
        print(f"Processing {self.currency} payment")
                print(f"Wallet:  {self.wallet_address[:6]}...{self.wallet_
address[-4:]}")
        print("Waiting for blockchain confirmation...")
        return super().process_payment()

    def send_confirmation(self):
        """Подтверждение через блокчейн"""
        print(f"Transaction recorded in {self.currency} blockchain")
# Полиморфное использование различных процессоров
def process_order_payments(processors):
    """Обработка списка платежей разными способами"""
    total_fees = 0
    successful_payments = 0

    for processor in processors:
        print(f"\n--- Processing payment #{successful_payments + 1} ---")
        if processor.process_payment():
```

```
            processor.send_confirmation()
            total_fees += processor.transaction_fee
            successful_payments += 1
        print("-" * 45)

    print(f"\nSummary:")
    print(f"Successful payments: {successful_payments}/{len(processors)}")
    print(f"Total fees: ${total_fees:.2f}")
# Создаем различные процессоры платежей
payments = [
    CreditCardProcessor(150.00, "1234567890123456", 123),
    PayPalProcessor(75.50, "user@example.com"),
    CryptocurrencyProcessor(200.00, "1A2B3C4D5E6F7G8H9I0J", "BTC"),
    CreditCardProcessor(25.00, "9876543210987654", 456)
]
# Одна функция обрабатывает все типы платежей
process_order_payments(payments)
```

Практическая польза полиморфизма

Полиморфизм решает несколько важных задач в программировании. Он **упрощает код** — одна функция может работать с множеством типов объектов, не зная об их конкретной реализации. Это делает код более **гибким** — новые типы можно добавлять без изменения существующего кода. Полиморфизм также **снижает связанность** между компонентами системы, поскольку код зависит от интерфейсов, а не от конкретных реализаций.

Рассмотрим финальный пример, демонстрирующий все преимущества полиморфизма:

```
class Notification:
    """Базовый интерфейс для уведомлений"""

    def send(self, recipient, message):
        raise NotImplementedError("Subclass must implement send method")
class EmailNotification(Notification):
    def send(self, recipient, message):
        print(f" Email sent to {recipient}: {message}")
class SMSNotification(Notification):
    def send(self, recipient, message):
        # Ограничиваем длину SMS
         short_message = message[:160] + "..." if len(message) > 160 else
message
        print(f" SMS sent to {recipient}: {short_message}")
class PushNotification(Notification):
    def send(self, recipient, message):
        print(f" Push notification to {recipient}: {message}")
class NotificationService:
    """Сервис для отправки уведомлений"""

    def __init__(self):
        self.providers = []
```

```
    def add_provider(self, provider):
        """Добавить провайдера уведомлений"""
        self.providers.append(provider)

    def send_notification(self, recipient, message, provider_type=None):
        """Отправить уведомление через указанного или всех провайдеров"""
        if provider_type:
            # Отправить через конкретного провайдера
            for provider in self.providers:
                if isinstance(provider, provider_type):
                    provider.send(recipient, message)
                    return
            print(f"Provider {provider_type.__name__} not found")
        else:
            # Отправить через всех провайдеров
            for provider in self.providers:
                provider.send(recipient, message)
# Использование полиморфизма в сервисе уведомлений
service = NotificationService()
service.add_provider(EmailNotification())
service.add_provider(SMSNotification())
service.add_provider(PushNotification())
# Отправка уведомлений - код не зависит от конкретных типов провайдеров
message = "Your order has been shipped and is on its way to you!"
print("Sending to all providers:")
service.send_notification("user@example.com", message)
print("\nSending only via email:")
service.send_notification("user@example.com", message, EmailNotification)
```

Полиморфизм — это не просто техническая особенность языка, а фундаментальный принцип проектирования, который позволяет создавать гибкие, расширяемые и поддерживаемые программы. Он работает рука об руку с наследованием и инкапсуляцией, образуя триаду основных принципов объектно-ориентированного программирования.

9.7. Специальные методы: магия Python

Специальные методы (special methods), также известные как **«магические методы» (magic methods)** или **«дандер-методы» (dunder methods — от double underscore)**, представляют собой одну из самых мощных особенностей Python. Эти методы начинаются и заканчиваются двойными подчеркиваниями и позволяют объектам ваших классов интегрироваться со встроенными функциями Python и операторами языка.

Специальные методы — это договор между вашими объектами и интерпретатором Python. Они определяют, как объект ведет себя в различных ситуациях: при создании, при выводе на экран, при сравнении с другими объектами, при использовании в циклах и многое другое. Благодаря им ваши объекты могут выглядеть и вести себя как встроенные типы данных Python.

Основы создания и представления объектов

Метод `__init__` уже хорошо знаком нам — это конструктор класса, который вызывается при создании нового экземпляра. Однако существуют и другие важные методы для работы с представлением объектов:

```
class Book:
    """Класс книги, демонстрирующий специальные методы"""

    def __init__(self, title, author, pages, price):
        self.title = title
        self.author = author
        self.pages = pages
        self.price = price
        self.isbn = None

    def __str__(self):
        """Человекочитаемое представление объекта"""
        return f'"{self.title}" by {self.author}'

    def __repr__(self):
        """Техническое представление объекта для разработчиков"""
        return f'Book("{self.title}", "{self.author}", {self.pages}, {self.
price})'

    def __len__(self):
        """Возвращает длину объекта (количество страниц)"""
        return self.pages
# Демонстрация работы специальных методов
book = Book("1984", "George Orwell", 328, 12.99)
print(str(book))    # "1984" by George Orwell
print(repr(book))   # Book("1984", "George Orwell", 328, 12.99)
print(len(book))    # 328
# При выводе объекта Python автоматически использует __str__
print(book)         # "1984" by George Orwell
# В интерактивном режиме и списках используется __repr__
books = [book]
print(books)        # [Book("1984", "George Orwell", 328, 12.99)]
```

Различие между `__str__` и `__repr__` принципиально важно: `__str__` должен возвращать красивое, понятное пользователю описание объекта, а `__repr__` — однозначное техническое представление, в идеале позволяющее воссоздать объект.

Операторы сравнения и арифметические операции

Специальные методы позволяют определить, как объекты вашего класса будут вести себя с операторами сравнения и арифметическими операциями:

```
class Money:
    """Класс для работы с денежными суммами"""

    def __init__(self, amount, currency="USD"):
```

```
        self.amount = amount
        self.currency = currency

    def __str__(self):
        return f"{self.amount:.2f} {self.currency}"

    def __repr__(self):
        return f"Money({self.amount}, '{self.currency}')"

    # Операторы сравнения
    def __eq__(self, other):
        """Равенство (==)"""
        if not isinstance(other, Money):
            return False
          return self.amount == other.amount and self.currency == other.
currency

    def __lt__(self, other):
        """Меньше (<)"""
        if not isinstance(other, Money):
            raise TypeError(f"Cannot compare Money with {type(other)}")
        if self.currency != other.currency:
            raise ValueError(f"Cannot compare different currencies: {self.
currency} and {other.currency}")
        return self.amount < other.amount

    def __le__(self, other):
        """Меньше или равно (<=)"""
        return self == other or self < other

    def __gt__(self, other):
        """Больше (>)"""
        if not isinstance(other, Money):
            raise TypeError(f"Cannot compare Money with {type(other)}")
        if self.currency != other.currency:
            raise ValueError(f"Cannot compare different currencies")
        return self.amount > other.amount

    def __ge__(self, other):
        """Больше или равно (>=)"""
        return self == other or self > other

    def __ne__(self, other):
        """Не равно (!=)"""
        return not self == other

    # Арифметические операции
    def __add__(self, other):
        """Сложение (+)"""
        if isinstance(other, Money):
            if self.currency != other.currency:
```

```
                raise ValueError(f"Cannot add different currencies: {self.
currency} and {other.currency}")
            return Money(self.amount + other.amount, self.currency)
        elif isinstance(other, (int, float)):
            return Money(self.amount + other, self.currency)
        else:
            raise TypeError(f"Cannot add Money and {type(other)}")

    def __sub__(self, other):
        """Вычитание (-)"""
        if isinstance(other, Money):
            if self.currency != other.currency:
                raise ValueError(f"Cannot subtract different currencies")
            return Money(self.amount - other.amount, self.currency)
        elif isinstance(other, (int, float)):
            return Money(self.amount - other, self.currency)
        else:
            raise TypeError(f"Cannot subtract {type(other)} from Money")

    def __mul__(self, other):
        """Умножение (*)"""
        if isinstance(other, (int, float)):
            return Money(self.amount * other, self.currency)
        else:
            raise TypeError(f"Cannot multiply Money by {type(other)}")

    def __truediv__(self, other):
        """Деление (/)"""
        if isinstance(other, (int, float)):
            if other == 0:
                raise ZeroDivisionError("Cannot divide by zero")
            return Money(self.amount / other, self.currency)
        else:
            raise TypeError(f"Cannot divide Money by {type(other)}")
# Демонстрация арифметических операций
price1 = Money(25.99)
price2 = Money(15.50)
tax_rate = Money(2.50)
print(price1)                    # 25.99 USD
print(price1 + price2)           # 41.49 USD
print(price1 - tax_rate)         # 23.49 USD
print(price1 * 2)                # 51.98 USD
print(price1 / 2)                # 12.99 USD
# Операторы сравнения
print(price1 > price2)           # True
print(price1 == Money(25.99))    # True
print(price1 != price2)          # True
# Сортировка работает автоматически благодаря операторам сравнения
prices = [Money(50), Money(25.99), Money(100), Money(10.50)]
sorted_prices = sorted(prices)
```

```
print([str(p) for p in sorted_prices])  # ['10.50 USD', '25.99 USD', '50.00
USD', '100.00 USD']
```

Контейнерные методы: работа с коллекциями

Специальные методы позволяют объектам вести себя как контейнеры — поддерживать индексацию, итерацию и проверку принадлежности:

```
class Playlist:
    """Класс плейлиста, ведущий себя как контейнер"""

    def __init__(self, name):
        self.name = name
        self.songs = []

    def __str__(self):
        return f"Playlist '{self.name}' with {len(self.songs)} songs"

    def __len__(self):
        """Длина плейлиста"""
        return len(self.songs)

    def __getitem__(self, index):
        """Доступ по индексу playlist[index]"""
        return self.songs[index]

    def __setitem__(self, index, value):
        """Установка значения по индексу playlist[index] = value"""
        self.songs[index] = value

    def __delitem__(self, index):
        """Удаление элемента del playlist[index]"""
        del self.songs[index]

    def __contains__(self, song):
        """Проверка принадлежности 'song' in playlist"""
        return song in self.songs

    def __iter__(self):
        """Итерация по плейлисту"""
        return iter(self.songs)

    def add_song(self, song):
        """Добавить песню в плейлист"""
        if song not in self.songs:
            self.songs.append(song)
            print(f"Added '{song}' to playlist '{self.name}'")
        else:
            print(f"'{song}' is already in the playlist")

    def remove_song(self, song):
```

```
        """Удалить песню из плейлиста"""
        if song in self.songs:
            self.songs.remove(song)
            print(f"Removed '{song}' from playlist '{self.name}'")
        else:
            print(f"'{song}' not found in playlist")
# Демонстрация контейнерных методов
my_playlist = Playlist("My Favorites")
# Добавляем песни
my_playlist.add_song("Bohemian Rhapsody")
my_playlist.add_song("Stairway to Heaven")
my_playlist.add_song("Hotel California")
print(my_playlist)              # Playlist 'My Favorites' with 3 songs
print(len(my_playlist))         # 3
# Доступ по индексу
print(my_playlist[0])           # Bohemian Rhapsody
my_playlist[1] = "Yesterday"    # Замена второй песни
# Проверка принадлежности
print("Yesterday" in my_playlist)        # True
print("Imagine" in my_playlist)          # False
# Итерация
print("All songs in playlist:")
for song in my_playlist:
    print(f"  - {song}")
# Удаление по индексу
del my_playlist[2]
print(f"After deletion: {len(my_playlist)} songs")
```

Продвинутые специальные методы

Существуют специальные методы для более сложных операций, таких как копирование объектов, вызов объекта как функции и управление атрибутами:

```
class Counter:
    """Счетчик с продвинутыми возможностями"""

    def __init__(self, initial_value=0, step=1):
        self.value = initial_value
        self.step = step
        self.history = [initial_value]

    def __str__(self):
        return f"Counter(value={self.value}, step={self.step})"

    def __call__(self, increment=None):
        """Позволяет вызывать объект как функцию"""
        if increment is None:
            increment = self.step

        old_value = self.value
        self.value += increment
```

```
        self.history.append(self.value)

        print(f"Counter incremented from {old_value} to {self.value}")
        return self.value

    def __copy__(self):
        """Поверхностное копирование"""
        new_counter = Counter(self.value, self.step)
        new_counter.history = self.history.copy()
        return new_counter

    def __bool__(self):
        """Логическое значение объекта"""
        return self.value != 0

    def __abs__(self):
        """Абсолютное значение abs(counter)"""
        return abs(self.value)

    def __neg__(self):
        """Унарный минус -counter"""
        new_counter = Counter(-self.value, self.step)
        new_counter.history = [-h for h in self.history]
        return new_counter
# Демонстрация продвинутых методов
import copy
counter = Counter(5, 2)
print(counter)          # Counter(value=5, step=2)
# Вызов объекта как функции
counter()               # Counter incremented from 5 to 7
counter(10)             # Counter incremented from 7 to 17
# Логическое значение
if counter:
    print("Counter is non-zero")
# Копирование
counter_copy = copy.copy(counter)
print(f"Original: {counter}")
print(f"Copy: {counter_copy}")
# Унарные операции
print(f"Absolute value: {abs(Counter(-10))}")  # 10
negative_counter = -counter
print(f"Negated: {negative_counter}")         # Counter(value=-17, step=2)
```

Специальные методы превращают ваши классы в полноценных «граждан» экосистемы Python. Они позволяют объектам естественно интегрироваться с встроенными функциями языка, операторами и синтаксическими конструкциями. Правильное использование магических методов делает код более интуитивным и «пайтоническим», позволяя пользователям ваших классов работать с ними так же естественно, как с встроенными типами данных.

9.8. Декораторы: модификация поведения

Декораторы (decorators) — это мощный механизм Python, позволяющий изменять или расширять поведение функций и методов без непосредственного изменения их кода. Декоратор — это функция, которая принимает другую функцию в качестве аргумента и возвращает модифицированную версию этой функции.

Концепция декораторов основана на том, что функции в Python являются объектами первого класса — их можно передавать как параметры, возвращать из других функций и присваивать переменным. Декораторы предоставляют элегантный синтаксис для применения таких модификаций с помощью символа @.

Встроенные декораторы классов

Начнем с изучения встроенных декораторов, которые часто используются в объектно-ориентированном программировании. Каждый из них решает определенные задачи и изменяет поведение методов класса.

Декоратор `@property`

Декоратор `@property` превращает метод в атрибут, позволяя использовать синтаксис доступа к атрибуту для выполнения вычислений или проверок:

```
class Temperature:
    """Класс для работы с температурой в разных шкалах"""

    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Температура в Цельсиях"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Установка температуры в Цельсиях с валидацией"""
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Температура в Фаренгейтах (только чтение)"""
        return self._celsius * 9/5 + 32

    @property
    def kelvin(self):
        """Температура в Кельвинах (только чтение)"""
        return self._celsius + 273.15

    def __str__(self):
```

```
            return f"{self.celsius:.1f}°C ({self.fahrenheit:.1f}°F, {self.
kelvin:.1f}K)"
# Использование свойств
temp = Temperature(25)
print(temp)                 # 25.0°C (77.0°F, 298.1K)
# Свойства используются как атрибуты
print(temp.celsius)         # 25
print(temp.fahrenheit)      # 77.0
print(temp.kelvin)          # 298.15
# Изменение через setter
temp.celsius = 0
print(temp)                 # 0.0°C (32.0°F, 273.1K)
try:
    temp.celsius = -300    # ValueError: Temperature cannot be below absolute
zero
except ValueError as e:
    print(f"Error: {e}")
```

Декоратор `@classmethod`

Декоратор `@classmethod` создает **метод класса**, который получает сам класс в качестве первого аргумента (обычно называемого `cls`) вместо экземпляра. Методы класса часто используются для создания альтернативных конструкторов:

```
class Person:
    """Класс человека с альтернативными конструкторами"""

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_string(cls, person_str):
        """Альтернативный конструктор из строки 'Name-Age'"""
        name, age = person_str.split('-')
        return cls(name, int(age))

    @classmethod
    def from_dict(cls, person_dict):
        """Альтернативный конструктор из словаря"""
        return cls(person_dict['name'], person_dict['age'])

    @classmethod
    def get_species(cls):
        """Метод класса для получения общей информации"""
        return "Homo sapiens"

    def __str__(self):
        return f"{self.name}, {self.age} years old"
# Обычное создание объекта
person1 = Person("Alice", 30)
print(person1)                      # Alice, 30 years old
```

```
# Альтернативные конструкторы
person2 = Person.from_string("Bob-25")
print(person2)                        # Bob, 25 years old
person_data = {"name": "Charlie", "age": 35}
person3 = Person.from_dict(person_data)
print(person3)                        # Charlie, 35 years old
# Метод класса
print(Person.get_species())           # Homo sapiens
print(person1.get_species())          # Тоже работает через экземпляр
```

Декоратор `@staticmethod`

Декоратор `@staticmethod` создает **статический метод**, который не получает ни экземпляр, ни класс в качестве первого аргумента. Это обычная функция, которая логически связана с классом, но не зависит от состояния класса или экземпляра:

```
class MathUtils:
    """Утилитарный класс с математическими функциями"""

    PI = 3.14159265359

    def __init__(self, precision=2):
        self.precision = precision

    @staticmethod
    def is_prime(n):
        """Проверка, является ли число простым"""
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True

    @staticmethod
    def factorial(n):
        """Вычисление факториала числа"""
        if n < 0:
          raise ValueError("Factorial is not defined for negative numbers")
        if n <= 1:
            return 1
        return n * MathUtils.factorial(n - 1)

    @staticmethod
    def distance(x1, y1, x2, y2):
        """Расчет расстояния между двумя точками"""
        return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

    @classmethod
    def circle_area(cls, radius):
        """Метод класса для расчета площади круга"""
        return cls.PI * radius ** 2
```

```
    def format_number(self, number):
        """Метод экземпляра для форматирования числа"""
        return round(number, self.precision)
# Использование различных типов методов
print(MathUtils.is_prime(17))           # True (статический метод)
print(MathUtils.factorial(5))           # 120 (статический метод)
print(MathUtils.distance(0, 0, 3, 4))   # 5.0 (статический метод)
print(MathUtils.circle_area(5))         # 78.53981633975 (метод класса)
math_util = MathUtils(precision=3)
print(math_util.format_number(3.14159)) # 3.142 (метод экземпляра)
# Статические методы можно вызывать и через экземпляр
print(math_util.is_prime(13))           # True
```

Создание пользовательских декораторов

Понимание встроенных декораторов подготавливает нас к созданию собственных. Декораторы особенно полезны для логирования, измерения производительности, проверки прав доступа и кеширования результатов:

```
def timer(func):
    """Декоратор для измерения времени выполнения функции"""
    import time
    import functools

    @functools.wraps(func)  # Сохраняет метаданные оригинальной функции
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
          print(f"{func.__name__} executed in {end_time - start_time:.4f}
seconds")
        return result
    return wrapper
def validate_positive(func):
    """Декоратор для проверки, что все числовые аргументы положительные"""
    import functools

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # Проверяем позиционные аргументы
        for arg in args:
            if isinstance(arg, (int, float)) and arg <= 0:
                raise ValueError(f"Argument {arg} must be positive")

        # Проверяем именованные аргументы
        for key, value in kwargs.items():
            if isinstance(value, (int, float)) and value <= 0:
               raise ValueError(f"Argument {key}={value} must be positive")
```

```
        return func(*args, **kwargs)
    return wrapper
def log_calls(func):
    """Декоратор для логирования вызовов методов"""
    import functools

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
           print(f"Calling {func.__name__} with args: {args[1:]}, kwargs:
{kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned: {result}")
        return result
    return wrapper
class Calculator:
    """Калькулятор с декорированными методами"""

    def __init__(self, name="Calculator"):
        self.name = name
        self.operations_count = 0

    @timer
    @validate_positive
    @log_calls
    def power(self, base, exponent):
        """Возведение в степень"""
        self.operations_count += 1
        result = base ** exponent
        return result

    @timer
    @log_calls
    def fibonacci(self, n):
        """Вычисление числа Фибоначчи (медленная рекурсивная версия)"""
        self.operations_count += 1
        if n <= 1:
            return n
        return self.fibonacci(n-1) + self.fibonacci(n-2)

    @validate_positive
    def divide(self, a, b):
        """Деление с проверкой положительности"""
        self.operations_count += 1
        return a / b
# Демонстрация работы пользовательских декораторов
calc = Calculator("Advanced Calculator")
print("=== Testing power method ===")
result = calc.power(2, 8)
print(f"Result: {result}\n")
print("=== Testing validation ===")
try:
```

```
    calc.divide(10, -5)  # ValueError из-за отрицательного числа
except ValueError as e:
    print(f"Validation error: {e}\n")
print("=== Testing fibonacci with timer ===")
fib_result = calc.fibonacci(10)
print(f"Fibonacci result: {fib_result}")
print(f"Total operations: {calc.operations_count}")
```

Декораторы для методов класса

Декораторы могут быть особенно полезны при работе с методами классов, обеспечивая кросс-сквозную функциональность:

```
def access_control(required_role):
    """Декоратор для контроля доступа к методам"""
    import functools

    def decorator(func):
        @functools.wraps(func)
        def wrapper(self, *args, **kwargs):
            if not hasattr(self, 'user_role'):
               raise AttributeError("Object must have user_role attribute")

            if self.user_role != required_role:
                  raise PermissionError(f"Access denied. Required role:
{required_role}")

            return func(self, *args, **kwargs)
        return wrapper
    return decorator
def audit_trail(func):
    """Декоратор для ведения журнала операций"""
    import functools
    import datetime

    @functools.wraps(func)
    def wrapper(self, *args, **kwargs):
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        operation = f"{func.__name__}({', '.join(map(str, args))})"

        # Выполняем операцию
        try:
            result = func(self, *args, **kwargs)
            status = "SUCCESS"
            self._audit_log.append(f"[{timestamp}] {operation} - {status}")
            return result
        except Exception as e:
            status = f"ERROR: {e}"
            self._audit_log.append(f"[{timestamp}] {operation} - {status}")
            raise
    return wrapper
```

```
class BankAccount:
    """Банковский счет с контролем доступа и аудитом"""

    def __init__(self, account_number, initial_balance, user_role="user"):
        self.account_number = account_number
        self.balance = initial_balance
        self.user_role = user_role
        self._audit_log = []

    @audit_trail
    def deposit(self, amount):
        """Пополнение счета"""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount
        return self.balance

    @audit_trail
    def withdraw(self, amount):
        """Снятие денег"""
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount
        return self.balance

    @access_control("admin")
    @audit_trail
    def set_balance(self, new_balance):
        """Прямая установка баланса (только для администраторов)"""
        old_balance = self.balance
        self.balance = new_balance
        return f"Balance changed from {old_balance} to {new_balance}"

    @access_control("admin")
    def view_audit_log(self):
        """Просмотр журнала операций (только для администраторов)"""
        return self._audit_log.copy()

    def get_balance(self):
        """Получить текущий баланс"""
        return self.balance
# Демонстрация декорированных методов класса
print("=== User account operations ===")
user_account = BankAccount("12345", 1000, "user")
user_account.deposit(500)
print(f"Balance after deposit: {user_account.get_balance()}")
user_account.withdraw(200)
print(f"Balance after withdrawal: {user_account.get_balance()}")
# Попытка выполнить операцию администратора
```

```
try:
    user_account.set_balance(5000)
except PermissionError as e:
    print(f"Access denied: {e}")
print("\n=== Admin account operations ===")
admin_account = BankAccount("67890", 2000, "admin")
admin_account.deposit(1000)
admin_result = admin_account.set_balance(5000)
print(admin_result)
# Просмотр журнала операций
audit_log = admin_account.view_audit_log()
print("\nAudit log:")
for entry in audit_log:
    print(f"  {entry}")
```

Композиция декораторов

Декораторы можно комбинировать, создавая сложные цепочки поведения. Важно помнить, что декораторы применяются снизу вверх:

```
class DataProcessor:
    """Класс обработки данных с множественными декораторами"""

    def __init__(self, name):
        self.name = name
        self._audit_log = []

    @timer                        # Применяется последним
    @cache_result(max_size=3)     # Применяется вторым
    @audit_trail                 # Применяется первым
    def complex_operation(self, data):
        """Сложная операция с множественными декораторами"""
        import time
        time.sleep(0.1)  # Имитация сложной обработки

        # Простая обработка данных
        processed = [x * 2 for x in data if x > 0]
        return sum(processed)
# Демонстрация композиции декораторов
processor = DataProcessor("Main Processor")
print("=== First call (no cache) ===")
result1 = processor.complex_operation([1, 2, 3, 4, 5])
print(f"Result: {result1}")
print("\n=== Second call (from cache) ===")
result2 = processor.complex_operation([1, 2, 3, 4, 5])
print(f"Result: {result2}")
print("\n=== Third call (different data) ===")
result3 = processor.complex_operation([2, 4, 6, 8])
print(f"Result: {result3}")
print(f"\nCache info: {processor.complex_operation.cache_info()}")
```

Декораторы представляют собой мощный инструмент для создания чистого, модульного и повторно используемого кода. Они позволяют отделить основную логику от вспомогательной функциональности, такой как логирование, кеширование, проверка прав доступа или измерение производительности.

Встроенные декораторы `@property`, `@classmethod` и `@staticmethod` решают частые задачи объектно-ориентированного программирования, делая код более выразительным и «пайтоническим». Пользовательские декораторы открывают неограниченные возможности для расширения и модификации поведения функций и методов, следуя принципу единственной ответственности и делая код более поддерживаемым.

9.9. Свойства: контролируемый доступ к атрибутам

Свойства (properties) в Python предоставляют механизм контролируемого доступа к атрибутам объекта. Они позволяют использовать простой синтаксис доступа к атрибутам (`obj.attribute`), скрывая при этом сложную логику валидации, вычислений или преобразований. Свойства представляют собой элегантное решение проблемы инкапсуляции, позволяя коду выглядеть просто, оставаясь при этом мощным и защищенным.

Основная идея свойств заключается в том, что они выглядят как обычные атрибуты для пользователя класса, но на самом деле являются методами. Это означает, что можно начать с простого атрибута и позже добавить к нему логику валидации или вычислений, не изменяя интерфейс класса.

Создание свойств с помощью функции `property()`

Функция `property()` — это классический способ создания свойств. Она принимает до четырех аргументов: **`getter`**, **`setter`**, **`deleter`** и **`doc`** (документацию):

```
class Circle:
    """Класс окружности с контролируемым радиусом"""

    def __init__(self, radius):
        self._radius = None
        self.radius = radius  # Используем setter для валидации

    def get_radius(self):
        """Getter для радиуса"""
        return self._radius

    def set_radius(self, value):
        """Setter для радиуса с валидацией"""
        if not isinstance(value, (int, float)):
            raise TypeError("Radius must be a number")
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = float(value)

    def del_radius(self):
        """Deleter для радиуса"""
```

```
        print("Deleting radius value")
        self._radius = None

    def get_area(self):
        """Вычисляемое свойство для площади"""
        if self._radius is None:
            return 0
        return 3.14159 * self._radius ** 2

    def get_circumference(self):
        """Вычисляемое свойство для длины окружности"""
        if self._radius is None:
            return 0
        return 2 * 3.14159 * self._radius

    # Создание свойств с помощью property()
     radius = property(get_radius, set_radius, del_radius, "Radius of the
circle")
    area = property(get_area, doc="Area of the circle")
     circumference = property(get_circumference, doc="Circumference of the
circle")
# Использование свойств
circle = Circle(5)
print(f"Radius: {circle.radius}")           # 5.0
print(f"Area: {circle.area:.2f}")           # 78.54
print(f"Circumference: {circle.circumference:.2f}")  # 31.42
# Изменение радиуса через setter
circle.radius = 10
print(f"New area: {circle.area:.2f}")       # 314.16
# Валидация работает
try:
    circle.radius = -5
except ValueError as e:
    print(f"Error: {e}")                   # Error: Radius must be positive
try:
    circle.radius = "invalid"
except TypeError as e:
    print(f"Error: {e}")                   # Error: Radius must be a number
# Удаление свойства
del circle.radius
print(f"Area after deletion: {circle.area}") # 0
```

Использование декоратора `@property`

Декораторы предоставляют более элегантный и читаемый синтаксис для создания свойств. Этот подход мы уже видели ранее, но рассмотрим более детально:

```
class BankAccount:
    """Банковский счет с контролируемым балансом"""

    def __init__(self, account_number, initial_balance=0):
```

```
        self.account_number = account_number
        self._balance = None
        self._transaction_history = []
        self.balance = initial_balance  # Используем setter

    @property
    def balance(self):
        """Текущий баланс счета"""
        return self._balance

    @balance.setter
    def balance(self, value):
        """Установка баланса с валидацией и логированием"""
        if not isinstance(value, (int, float)):
            raise TypeError("Balance must be a number")
        if value < 0:
            raise ValueError("Balance cannot be negative")

        old_balance = self._balance
        self._balance = float(value)

        # Логируем изменение
        if old_balance is not None:
            change = value - old_balance
            transaction_type = "deposit" if change > 0 else "withdrawal"
            self._transaction_history.append({
                'type': transaction_type,
                'amount': abs(change),
                'new_balance': self._balance
            })

    @property
    def transaction_count(self):
        """Количество транзакций (только для чтения)"""
        return len(self._transaction_history)

    @property
    def last_transaction(self):
        """Последняя транзакция (только для чтения)"""
        if self._transaction_history:
            return self._transaction_history[-1].copy()
        return None

    @property
    def account_status(self):
        """Статус счета на основе баланса"""
        if self._balance is None:
            return "uninitialized"
        elif self._balance == 0:
            return "zero_balance"
        elif self._balance < 100:
```

```
            return "low_balance"
        else:
            return "active"

    def deposit(self, amount):
        """Пополнение счета"""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance = self._balance + amount

    def withdraw(self, amount):
        """Снятие средств"""
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self._balance:
            raise ValueError("Insufficient funds")
        self.balance = self._balance - amount
# Демонстрация работы свойств
account = BankAccount("ACC-001", 500)
print(f"Initial balance: ${account.balance}")
print(f"Account status: {account.account_status}")
# Операции со счетом
account.deposit(200)
print(f"After deposit: ${account.balance}")
print(f"Transaction count: {account.transaction_count}")
print(f"Last transaction: {account.last_transaction}")
account.withdraw(150)
print(f"After withdrawal: ${account.balance}")
print(f"Account status: {account.account_status}")
# Попытка прямой установки неправильного значения
try:
    account.balance = -100
except ValueError as e:
    print(f"Validation error: {e}")
# Свойства только для чтения нельзя изменить
try:
    account.transaction_count = 10
except AttributeError as e:
    print(f"Read-only property error: {e}")
```

Свойства с `deleter`

Иногда нужно контролировать не только получение и установку значений, но и их удаление. Для этого используется **`deleter`**:

```
class ConfigurationManager:
    """Менеджер конфигурации с контролируемым удалением"""

    def __init__(self):
        self._config = {}
        self._required_keys = {'database_url', 'api_key', 'debug_mode'}
```

```
    @property
    def database_url(self):
        """URL базы данных"""
        return self._config.get('database_url')

    @database_url.setter
    def database_url(self, value):
        """Установка URL базы данных с валидацией"""
        if not isinstance(value, str):
            raise TypeError("Database URL must be a string")
       if not value.startswith(('postgresql://', 'mysql://', 'sqlite://')):
            raise ValueError("Invalid database URL format")
        self._config['database_url'] = value

    @database_url.deleter
    def database_url(self):
        """Удаление URL базы данных с проверкой"""
        if 'database_url' in self._config:
            print("Warning: Removing database URL from configuration")
            del self._config['database_url']
        else:
            print("Database URL is not set")

    @property
    def api_key(self):
        """API ключ"""
        return self._config.get('api_key', '***hidden***')

    @api_key.setter
    def api_key(self, value):
        """Установка API ключа"""
        if not isinstance(value, str):
            raise TypeError("API key must be a string")
        if len(value) < 16:
            raise ValueError("API key too short (minimum 16 characters)")
        self._config['api_key'] = value

    @api_key.deleter
    def api_key(self):
        """Безопасное удаление API ключа"""
        if 'api_key' in self._config:
            # Обнуляем значение для безопасности
            self._config['api_key'] = "0" * len(self._config['api_key'])
            del self._config['api_key']
            print("API key securely deleted")

    @property
    def is_complete(self):
        """Проверка полноты конфигурации"""
        return all(key in self._config for key in self._required_keys)
```

```
    def validate_configuration(self):
        """Валидация всей конфигурации"""
        missing_keys = self._required_keys - set(self._config.keys())
        if missing_keys:
          raise ValueError(f"Missing required configuration keys: {missing_
keys}")
        return True
# Использование свойств с deleter
config = ConfigurationManager()
# Установка конфигурации
config.database_url = "postgresql://localhost:5432/mydb"
config.api_key = "super_secret_key_123456"
config._config['debug_mode'] = True
print(f"Database URL: {config.database_url}")
print(f"API Key: {config.api_key}")
print(f"Configuration complete: {config.is_complete}")
# Удаление с контролем
del config.api_key
print(f"After API key deletion: {config.api_key}")
del config.database_url
print(f"Configuration complete after deletion: {config.is_complete}")
```

Свойства обеспечивают баланс между простотой использования и контролем. Они позволяют сохранять чистый и понятный интерфейс класса, скрывая сложности реализации. Это особенно важно при создании библиотек и API, где изменение внутренней реализации не должно затрагивать код пользователей.

9.10. Упражнения для закрепления пройденного материала

Объектно-ориентированное программирование представляет собой фундаментальную смену парадигмы мышления. Чтобы по-настоящему освоить ООП, недостаточно просто запомнить синтаксис — необходимо научиться видеть мир через призму объектов, их взаимодействий и иерархий. Предложенные упражнения построены по принципу постепенного усложнения и охватывают все аспекты объектно-ориентированного программирования от базовых концепций до продвинутых техник.

Упражнение 1. Моделирование банковского счета

Задача: создайте класс BankAccount, который моделирует банковский счет с базовой функциональностью.

Требования:

- Атрибуты: номер счета, имя владельца, текущий баланс.
- Методы: пополнение счета, снятие средств, получение баланса.
- Контроль: запрет снятия суммы, превышающей баланс.

Цель упражнения: понять базовые принципы создания классов, инициализации объектов и реализации методов. Осознать, как объекты инкапсулируют данные и поведение.

Критерии качества решения:

- Корректная обработка некорректных операций (отрицательные суммы, недостаток средств).
- Информативные сообщения об ошибках.
- Логичная структура кода.

Упражнение 2. Система управления книгами

Задача: разработайте класс Book для системы управления библиотекой.

Требования:

- Атрибуты: название, автор, год издания, ISBN, статус (доступна/выдана).
- Методы: выдача книги, возврат книги, получение информации о книге.
- Валидация: корректность ISBN, года издания.
- Развитие задачи: добавьте класс Library, который управляет коллекцией книг.
- Добавление/удаление книг.
- Поиск по автору, названию, году.
- Статистика: общее количество книг, количество выданных книг.

Цель упражнения: освоить принципы композиции объектов и взаимодействия между классами. Понять важность валидации данных в методах.

Упражнение 3. Умный термостат

Задача: создайте класс SmartThermostat, демонстрирующий принципы инкапсуляции.

Требования:

- Приватные атрибуты: текущая температура, целевая температура, режим работы.
- Свойства (property) для контролируемого доступа к атрибутам.
- Валидация: температура в разумных пределах (-50°C до +60°C).
- Методы: включение/выключение, изменение режима, получение статуса.

Аналитическое задание: объясните, почему прямой доступ к атрибутам температуры может быть опасным в реальной системе умного дома. Как инкапсуляция помогает решить эти проблемы?

Упражнение 4. Иерархия транспортных средств

Задача: создайте иерархию классов для моделирования различных транспортных средств.

Базовый класс `Vehicle`:

- Общие атрибуты: марка, модель, год выпуска, текущая скорость.
- Абстрактные методы: `start_engine()`, `stop_engine()`, `accelerate()`.
- Общие методы: `get_info()`, `set_speed()`.

Производные классы:

- `Car`: дополнительно количество дверей, тип коробки передач.
- `Motorcycle`: тип мотоцикла (спортивный, круизер, туристический).
- `Truck`: грузоподъемность, тип груза.

Практическое применение: создайте функцию `calculate_trip_cost()`, которая принимает список различных транспортных средств и рассчитывает стоимость поездки для каждого, демонстрируя полиморфизм.

Аналитическое задание: объясните, как принцип подстановки Лисков применяется к вашей иерархии классов. Приведите примеры нарушения этого принципа.

Упражнение 5. Умная коллекция

Задача: создайте класс `SmartList`, который расширяет возможности обычного списка.

Специальные возможности:

- Автоматическое логирование операций изменения.
- Ограничение максимального размера.
- Типизация элементов (только определенный тип).
- Статистика использования (количество добавлений, удалений, обращений).

Специальные методы:

- Все методы обычного списка.
- `__contains__`: поиск с подсчётом обращений.
- `__setitem__`: валидация типа при установке.
- `__delitem__`: логирование удаления.
- Context manager (`__enter__`, `__exit__`): временная блокировка изменений.

Цель упражнения: понять мощь специальных методов для создания объектов, неотличимых от встроенных типов Python.

Глава 10. Отладка и тестирование

10.1. Искусство отладки: стратегии поиска ошибок

Отладка (debugging) — это процесс поиска и исправления ошибок в программном коде. Этот навык часто определяет разницу между начинающим программистом и опытным разработчиком. Хорошие программисты не те, кто не делает ошибок — это те, кто умеет их быстро находить и исправлять.

Название debugging происходит от истории, когда Грейс Хоппер в 1947 году обнаружила настоящего жука (bug), застрявшего в реле компьютера Harvard Mark II. С тех пор термин «баг» стал обозначать любую ошибку в программе, а «отладка» — процесс их устранения.

Философия отладки

Отладка — это не случайный процесс проб и ошибок, а систематическая деятельность, основанная на логическом мышлении. Хороший отладчик подходит к проблеме как детектив: собирает улики, выдвигает гипотезы, проверяет их и делает выводы.

Основные принципы эффективной отладки включают понимание проблемы перед попыткой ее решить, систематический подход вместо хаотичных изменений, документирование найденных решений для будущего использования и, главное, сохранение спокойствия — эмоции мешают логическому мышлению.

Стратегии поиска ошибок

Стратегия «Разделяй и властвуй»

Эта стратегия заключается в постепенном сужении области поиска ошибки. Если программа состоит из 100 строк, не нужно проверять каждую — достаточно разделить код пополам и определить, в какой половине находится проблема.

```
def calculate_total_price(items, tax_rate, discount):
    """Вычисляет общую стоимость с учетом налога и скидки"""
    # Этап 1: Базовые вычисления
    subtotal = sum(item['price'] * item['quantity'] for item in items)

    # Этап 2: Применение скидки
    discounted_price = subtotal * (1 - discount)

    # Этап 3: Добавление налога
    total_with_tax = discounted_price * (1 + tax_rate)

    return total_with_tax
# Тестовые данные
items = [
    {'name': 'laptop', 'price': 1000, 'quantity': 1},
    {'name': 'mouse', 'price': 50, 'quantity': 2}
```

```
]
result = calculate_total_price(items, 0.1, 0.15)
print(f"Total price: {result}")
Если результат неверный, проверяем каждый этап отдельно:
def calculate_total_price_debug(items, tax_rate, discount):
    """Версия функции с промежуточной отладкой"""
    print("=== DEBUG: Calculating total price ===")

    # Проверяем входные данные
    print(f"Items: {items}")
    print(f"Tax rate: {tax_rate}")
    print(f"Discount: {discount}")

    # Этап 1: Базовые вычисления
    subtotal = sum(item['price'] * item['quantity'] for item in items)
    print(f"Subtotal: {subtotal}")

    # Этап 2: Применение скидки
    discounted_price = subtotal * (1 - discount)
    print(f"After discount: {discounted_price}")

    # Этап 3: Добавление налога
    total_with_tax = discounted_price * (1 + tax_rate)
    print(f"After tax: {total_with_tax}")

    return total_with_tax
```

Стратегия «От простого к сложному»

Начинайте отладку с максимально упрощенной версии проблемы. Если сложный алгоритм работает неправильно, создайте минимальный пример, который воспроизводит ошибку:

```
# Сложная функция с ошибкой
def analyze_sales_data(sales_records, filters):
    """Анализирует данные о продажах с различными фильтрами"""
    filtered_records = []

    for record in sales_records:
        if apply_filters(record, filters):
            filtered_records.append(transform_record(record))

    return aggregate_results(filtered_records)
# Упрощенная версия для отладки
def simple_sales_analysis():
    """Минимальный пример для проверки логики"""
    # Один простой случай
    record = {'date': '2023-01-01', 'amount': 100, 'product': 'laptop'}

    # Проверяем каждый шаг
    print(f"Original record: {record}")
```

```
    # Здесь проверяем каждую функцию отдельно
    filter_result = apply_filters(record, {'min_amount': 50})
    print(f"Filter result: {filter_result}")

    if filter_result:
        transformed = transform_record(record)
        print(f"Transformed: {transformed}")
```

Чтение и интерпретация трейсбеков

Трейсбек (traceback) — это отчет Python о том, что пошло не так в вашей программе. Умение читать трейсбеки — ключевой навык для любого Python-разработчика.

Анатомия трейсбека

Рассмотрим типичный пример ошибки:

```
def divide_numbers(a, b):
    """Делит два числа"""
    return a / b
def calculate_average(numbers):
    """Вычисляет среднее арифметическое"""
    total = sum(numbers)
    count = len(numbers)
    return divide_numbers(total, count)
def process_data():
    """Обрабатывает данные пользователя"""
    data = []  # Пустой список - источник ошибки!
    average = calculate_average(data)
    print(f"Average: {average}")
# Запускаем и получаем ошибку
process_data()
```

Этот код вызовет следующий трейсбек:

```
Traceback (most recent call last):  File "example.py", line 17, in <module>
process_data()  File "example.py", line 13, in process_data    average =
calculate_average(data)  File "example.py", line 8, in calculate_average
return divide_numbers(total, count)  File "example.py", line 3, in divide_
numbers    return a / bZeroDivisionError: division by zero
```

Чтение трейсбека снизу вверх

Трейсбек читается снизу вверх. Самая важная информация находится в конце:

1. **Тип ошибки:** `ZeroDivisionError`
2. **Описание ошибки:** `division by zero`
3. **Место возникновения:** функция `divide_numbers`, строка `return a / b`

Затем идем вверх по цепочке вызовов, чтобы понять, как мы дошли до этой ошибки:

1. `divide_numbers` была вызвана из `calculate_average`
2. `calculate_average` была вызвана из `process_data`
3. `process_data` была вызвана в основном коде

Типичные ошибки и их интерпретация

```
# 1. NameError - использование неопределенной переменной
def demo_name_error():
     print(undefined_variable)  # NameError: name 'undefined_variable' is
not defined
# 2. TypeError - неправильный тип данных
def demo_type_error():
    result = "5" + 3  # TypeError: can only concatenate str (not "int") to str
# 3. IndexError - выход за границы списка
def demo_index_error():
    numbers = [1, 2, 3]
    print(numbers[5])  # IndexError: list index out of range
# 4. KeyError - отсутствующий ключ в словаре
def demo_key_error():
    data = {'name': 'John'}
    print(data['age'])  # KeyError: 'age'
# 5. AttributeError - отсутствующий атрибут или метод
def demo_attribute_error():
    text = "hello"
    text.append(" world")  # AttributeError: 'str' object has no attribute
'append'
```

Print-отладка: простой и эффективный метод

Print-отладка — это добавление операторов `print()` в код для отслеживания выполнения программы и состояния переменных. Несмотря на кажущуюся примитивность, этот метод остается одним из самых популярных среди разработчиков.

Основы print-отладки

```
def find_maximum(numbers):
    """Находит максимальное число в списке"""
    print(f"DEBUG: Input numbers: {numbers}")  # Проверяем входные данные

    if not numbers:
        print("DEBUG: Empty list detected")  # Отслеживаем особые случаи
        return None

    max_num = numbers[0]
    print(f"DEBUG: Initial max_num: {max_num}")  # Начальное значение

    for i, num in enumerate(numbers[1:], 1):
       print(f"DEBUG: Step {i}, checking {num} vs {max_num}")  # Каждый шаг

        if num > max_num:
            max_num = num
            print(f"DEBUG: New maximum found: {max_num}")

    print(f"DEBUG: Final result: {max_num}")  # Финальный результат
    return max_num
# Тестирование
```

```
test_numbers = [3, 7, 2, 9, 1, 5]
result = find_maximum(test_numbers)
```

Условная отладка

Не всегда нужно видеть все отладочные сообщения. Создайте систему условной отладки:

```
DEBUG_MODE = True  # Глобальный флаг отладки
def debug_print(*args, **kwargs):
    """Выводит сообщение только в режиме отладки"""
    if DEBUG_MODE:
        print("[DEBUG]", *args, **kwargs)
def process_user_data(users):
    """Обрабатывает данные пользователей"""
    debug_print(f"Processing {len(users)} users")

    results = []
    for i, user in enumerate(users):
       debug_print(f"Processing user {i+1}: {user.get('name', 'Unknown')}")

        # Проверяем обязательные поля
        if 'email' not in user:
            debug_print(f"Warning: User {user.get('name')} has no email")
            continue

        # Обрабатываем пользователя
        processed_user = {
            'id': user.get('id'),
            'name': user.get('name'),
            'email': user['email'].lower(),
            'status': 'active'
        }

        results.append(processed_user)
        debug_print(f"User processed successfully: {processed_user}")

    debug_print(f"Total processed: {len(results)} users")
    return results
# Тестовые данные
test_users = [
    {'id': 1, 'name': 'Alice', 'email': 'ALICE@EXAMPLE.COM'},
    {'id': 2, 'name': 'Bob'},  # Нет email - вызовет предупреждение
    {'id': 3, 'name': 'Charlie', 'email': 'charlie@test.org'}
]
processed = process_user_data(test_users)
```

Эффективная отладка требует терпения, логического мышления и систематического подхода. Помните: каждая найденная и исправленная ошибка делает вас более опытным программистом. В следующем разделе мы изучим более продвинутые инструменты отладки, которые позволят вам работать еще более эффективно.

10.2. Отладчик Python: пошаговое выполнение

Хотя `print`-отладка остается популярным методом, настоящая мощь Python раскрывается при использовании специализированных инструментов отладки. **Отладчик (debugger)** — это программа, которая позволяет контролировать выполнение другой программы: останавливать ее в определенных местах, просматривать состояние переменных, выполнять код пошагово и анализировать стек вызовов.

Python предоставляет встроенный отладчик `pdb` (Python Debugger), а современные интегрированные среды разработки (IDE) предлагают графические интерфейсы для отладки. Овладение этими инструментами значительно повышает эффективность поиска и исправления ошибок.

Введение в pdb

Модуль `pdb` — это интерактивный отладчик Python, который позволяет исследовать программу во время ее выполнения. Он особенно полезен для понимания сложной логики программы и поиска неочевидных ошибок.

Основные способы запуска pdb

```
import pdb
def calculate_factorial(n):
    """Вычисляет факториал числа"""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

    # Устанавливаем точку останова
    pdb.set_trace()  # Выполнение остановится здесь

    result = 1
    for i in range(1, n + 1):
        result *= i
        print(f"Step {i}: result = {result}")

    return result
# Запуск функции
number = 5
factorial = calculate_factorial(number)
print(f"Factorial of {number} is {factorial}")
```

Альтернативный способ — запуск скрипта через командную строку:

```
python -m pdb script.py
```

...или с помощью `breakpoint()` (начиная с Python 3.7):

```
def analyze_text(text):
    """Анализирует текст и возвращает статистику"""
    words = text.split()

    # Современный способ установки точки останова
    breakpoint()  # Эквивалентно pdb.set_trace()
```

```
    word_count = len(words)
    char_count = len(text)
    avg_word_length = sum(len(word) for word in words) / word_count if words
else 0

    return {
        'word_count': word_count,
        'char_count': char_count,
        'avg_word_length': avg_word_length
    }
# Тестирование
sample_text = "Python is a powerful programming language"
stats = analyze_text(sample_text)
```

Основные команды pdb

Когда программа останавливается в отладчике, доступны следующие команды:

- n (next) — выполнить следующую строку;
- s (step) — шагнуть в функцию (если текущая строка содержит вызов функции);
- c (continue) — продолжить выполнение до следующей точки останова;
- l (list) — показать текущий код;
- p <variable> — вывести значение переменной;
- pp <variable> — красиво вывести значение переменной;
- w (where) — показать текущий стек вызовов;
- u (up) — подняться вверх по стеку;
- d (down) — опуститься вниз по стеку;
- q (quit) — выйти из отладчика.

Условные точки останова

Иногда нужно остановить выполнение программы только при определенных условиях:

```
def process_orders(orders):
    """Обрабатывает список заказов"""
    processed_orders = []

    for i, order in enumerate(orders):
        # Останавливаемся только для проблемных заказов
        if order.get('amount', 0) > 10000:
            breakpoint()  # Остановимся для больших заказов

        # Проверяем валидность заказа
        if 'customer_id' not in order or 'amount' not in order:
            print(f"Invalid order at index {i}: {order}")
            continue

        # Обрабатываем заказ
        processed_order = {
```

```
            'id': order.get('id', f'order_{i}'),
            'customer_id': order['customer_id'],
            'amount': order['amount'],
            'status': 'processed',
            'processed_at': time.time()
        }

        processed_orders.append(processed_order)

    return processed_orders
# Тестовые данные
import time
test_orders = [
    {'id': '001', 'customer_id': 'C001', 'amount': 150},
    {'id': '002', 'customer_id': 'C002', 'amount': 15000},  # Большой заказ
    {'customer_id': 'C003'},  # Невалидный заказ
    {'id': '004', 'customer_id': 'C004', 'amount': 75}
]
processed = process_orders(test_orders)
```

Отладка в IDE

Современные интегрированные среды разработки предоставляют графические интерфейсы для отладки, которые значительно упрощают процесс.

Возможности IDE-отладчиков
Графические отладчики обычно предоставляют:

- **Визуальные точки останова** — клик по номеру строки.
- **Панель переменных** — автоматический просмотр всех переменных.
- **Стек вызовов** — навигация по цепочке вызовов функций.
- **Консоль выражений** — вычисление выражений в контексте программы.
- **Пошаговое выполнение** — кнопки для различных режимов выполнения.

Создание отладочной конфигурации

```
class StudentGradeManager:
    """Система управления оценками студентов"""

    def __init__(self):
        self.students = {}

    def add_student(self, student_id, name):
        """Добавляет нового студента"""
        if student_id in self.students:
            raise ValueError(f"Student {student_id} already exists")

        self.students[student_id] = {
            'name': name,
            'grades': [],
            'average': 0.0
        }
```

```
    def add_grade(self, student_id, subject, grade):
        """Добавляет оценку студенту"""
        if student_id not in self.students:
            raise ValueError(f"Student {student_id} not found")

        if not (0 <= grade <= 100):
            raise ValueError("Grade must be between 0 and 100")

        # Хорошее место для точки останова в IDE
        grade_entry = {
            'subject': subject,
            'grade': grade,
            'timestamp': time.time()
        }

        self.students[student_id]['grades'].append(grade_entry)
        self._update_average(student_id)

    def _update_average(self, student_id):
        """Обновляет средний балл студента"""
        student = self.students[student_id]
        grades = [g['grade'] for g in student['grades']]

        if grades:
            student['average'] = sum(grades) / len(grades)
        else:
            student['average'] = 0.0

    def get_student_report(self, student_id):
        """Получает отчет по студенту"""
        if student_id not in self.students:
            return None

        student = self.students[student_id]

        # Еще одна хорошая точка для отладки
        report = {
            'name': student['name'],
            'total_grades': len(student['grades']),
            'average': round(student['average'], 2),
          'letter_grade': self._calculate_letter_grade(student['average']),
            'subjects': list(set(g['subject'] for g in student['grades']))
        }

        return report

    def _calculate_letter_grade(self, average):
        """Вычисляет буквенную оценку"""
        if average >= 90:
            return 'A'
```

```
        elif average >= 80:
            return 'B'
        elif average >= 70:
            return 'C'
        elif average >= 60:
            return 'D'
        else:
            return 'F'
# Демонстрационный код для отладки
def main():
    """Главная функция для тестирования системы"""
    manager = StudentGradeManager()

    # Добавляем студентов
    manager.add_student("S001", "Alice Johnson")
    manager.add_student("S002", "Bob Smith")
    manager.add_student("S003", "Charlie Brown")

    # Добавляем оценки
    grades_data = [
        ("S001", "Mathematics", 95),
        ("S001", "Physics", 87),
        ("S001", "Chemistry", 92),
        ("S002", "Mathematics", 78),
        ("S002", "Physics", 85),
        ("S003", "Mathematics", 65),
        ("S003", "Physics", 70),
        ("S003", "Chemistry", 68)
    ]

    for student_id, subject, grade in grades_data:
        manager.add_grade(student_id, subject, grade)

    # Генерируем отчеты
    for student_id in ["S001", "S002", "S003"]:
        report = manager.get_student_report(student_id)
        print(f"\n=== Report for {report['name']} ===")
        print(f"Average: {report['average']} ({report['letter_grade']})")
        print(f"Subjects: {', '.join(report['subjects'])}")
        print(f"Total grades: {report['total_grades']}")
if __name__ == "__main__":
    main()
```

Инспекция переменных и состояния программы

Одно из главных преимуществ отладчиков — возможность детального исследования состояния программы в любой момент выполнения.

Техники инспекции

```
def complex_data_processing(raw_data):
```

```
    """Обрабатывает сложную структуру данных"""

    # Начальная обработка
    normalized_data = []

    for item in raw_data:
        breakpoint()  # Остановимся здесь для исследования

        # В отладчике можно выполнить:
        # p item - посмотреть текущий элемент
        # p type(item) - проверить тип
        # p len(raw_data) - размер данных

        if isinstance(item, dict):
            # Нормализуем словарь
            normalized_item = {
                key.lower().replace(' ', '_'): value
                for key, value in item.items()
                if value is not None
            }
        elif isinstance(item, (list, tuple)):
            # Обрабатываем последовательность
            normalized_item = {
                f'field_{i}': value
                for i, value in enumerate(item)
                if value is not None
            }
        else:
            # Простое значение
            normalized_item = {'value': item}

        # Добавляем метаданные
        normalized_item['_original_type'] = type(item).__name__
        normalized_item['_processed_at'] = time.time()

        normalized_data.append(normalized_item)

    return normalized_data
# Тестовые данные различных типов
mixed_data = [
    {'Name': 'Alice', 'Age': 30, 'City': 'New York'},
    ['Bob', 25, 'Chicago'],
    {'PRODUCT': 'Laptop', 'Price': 999.99, 'Stock': None},
    'Simple string value',
    (1, 2, 3, 4, 5),
    {'': 'empty_key', 'normal_key': 'value'}
]
result = complex_data_processing(mixed_data)
# В отладчике можно исследовать результат:
for item in result:
    print(f"Processed item: {item}")
```

Отладка рекурсивных функций

Рекурсивные функции особенно сложны для отладки без специальных инструментов:

```
def binary_search_debug(arr, target, left=0, right=None, depth=0):
    """Бинарный поиск с отладочной информацией"""

    if right is None:
        right = len(arr) - 1

    # Отладочная точка для анализа рекурсии
    breakpoint()

    # В отладчике можно проанализировать:
    # p depth - глубина рекурсии
    # p left, right - границы поиска
    # p arr[left:right+1] - текущий подмассив

      print(f"{'  ' * depth}Depth {depth}: searching in range [{left},
{right}]")
    print(f"{'  ' * depth}Subarray: {arr[left:right+1]}")

    # Базовый случай
    if left > right:
        print(f"{'  ' * depth}Target {target} not found")
        return -1

    # Находим средний элемент
    mid = (left + right) // 2
    mid_value = arr[mid]

   print(f"{'  ' * depth}Checking middle element: arr[{mid}] = {mid_value}")

    if mid_value == target:
        print(f"{'  ' * depth}Found target at index {mid}!")
        return mid
    elif mid_value < target:
        print(f"{'  ' * depth}Target is greater, searching right half")
        return binary_search_debug(arr, target, mid + 1, right, depth + 1)
    else:
        print(f"{'  ' * depth}Target is smaller, searching left half")
        return binary_search_debug(arr, target, left, mid - 1, depth + 1)
# Тестирование рекурсивной отладки
sorted_array = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
target_value = 11
print(f"Searching for {target_value} in {sorted_array}")
result_index = binary_search_debug(sorted_array, target_value)
if result_index != -1:
    print(f"\nTarget found at index {result_index}")
else:
    print(f"\nTarget {target_value} not found in array")
```

Заключение по отладке

Мастерство отладки приходит с опытом, но знание правильных инструментов и техник значительно ускоряет этот процесс. Встроенный отладчик pdb предоставляет мощные возможности для исследования программы во время выполнения, а графические отладчики в IDE делают этот процесс более интуитивным и визуальным.

Ключевые принципы эффективной отладки включают систематический подход к поиску ошибок, использование условных точек останова для сложных сценариев, тщательное исследование состояния переменных в критических точках программы и документирование найденных проблем и их решений.

Помните: хороший отладчик не тот, кто быстро находит очевидные ошибки, а тот, кто может методично разобраться в сложной логике программы и найти неочевидные причины проблем. В следующем разделе мы изучим систему логирования — инструмент, который поможет не только отлаживать программы, но и отслеживать их работу в продакшене.

10.3. Логирование: правильный способ отслеживания работы программы

Логирование (logging) — это систематический способ записи событий, происходящих во время работы программы. В отличие от отладочных `print()` операторов, которые мы используем временно, система логирования предназначена для постоянного использования и предоставляет гибкие возможности контроля над тем, что, где и как записывается.

Представьте себе черный ящик самолета — это устройство записывает важные события полета, чтобы в случае необходимости можно было восстановить картину происходившего. Система логирования выполняет аналогичную функцию для программ: она создает постоянную запись работы приложения, которая помогает понять, что происходило в критические моменты.

Философия логирования

Логирование решает несколько важных задач. В процессе разработки оно помогает отслеживать выполнение программы без необходимости останавливать ее в отладчике. В продакшене логирование становится основным инструментом мониторинга работы приложения и диагностики проблем.

Хорошая система логирования должна быть информативной, но не избыточной, структурированной и легко читаемой, настраиваемой под разные условия работы и производительной — не замедлять работу программы значительно.

Модуль logging: основы

Python предоставляет мощный встроенный модуль `logging`, который покрывает большинство потребностей в логировании. Начнем с простейшего примера:

```
import logging
# Простейший пример логирования
logging.debug("This is a debug message")
logging.info("This is an info message")
```

```
logging.warning("This is a warning message")
logging.error("This is an error message")
logging.critical("This is a critical message")
```

По умолчанию будут выведены только сообщения уровня `WARNING` и выше. Это происходит потому, что уровень логирования по умолчанию установлен в `WARNING`.

Давайте создадим более практичный пример:

```
import logging
def calculate_division(a, b):
    """Функция деления с логированием"""
    logging.info(f"Starting division calculation: {a} / {b}")

    try:
        if b == 0:
            logging.error("Division by zero attempted!")
            raise ValueError("Cannot divide by zero")

        result = a / b
        logging.info(f"Division successful: {a} / {b} = {result}")
        return result

    except Exception as e:
        logging.critical(f"Unexpected error in division: {e}")
        raise
# Настроим базовый уровень логирования
logging.basicConfig(level=logging.INFO)
# Тестирование
try:
    result1 = calculate_division(10, 2)
    result2 = calculate_division(10, 0)  # Это вызовет ошибку
except ValueError as e:
    print(f"Caught error: {e}")
```

Уровни логирования

Система логирования Python использует пять стандартных уровней, каждый из которых имеет свое предназначение:

- **`DEBUG (10)`**
  Самый детальный уровень, используется для отладочной информации. Обычно интересен только при диагностике проблем.
- **`INFO (20)`**
  Общая информация о работе программы. Подтверждение того, что программа работает как ожидается.
- **`WARNING (30)`**
  Предупреждение о потенциальной проблеме. Программа продолжает работать, но что-то может пойти не так.
- **`ERROR (40)`**
  Серьезная проблема. Программа не смогла выполнить некоторую функцию.

- **CRITICAL (50)**
  Очень серьезная ошибка. Программа может прекратить работу.

Конфигурация логирования

Базовая конфигурация через `basicConfig()` подходит для простых случаев, но для реальных приложений нужны более гибкие настройки.

Продвинутая конфигурация

```
import logging
import sys
from datetime import datetime
def setup_advanced_logging():
    """Настройка продвинутого логирования"""

    # Создаем логгер
    logger = logging.getLogger('MyApp')
    logger.setLevel(logging.DEBUG)

    # Создаем форматтер
    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(filename)s:%(lineno)d
- %(message)s'
    )

    # Обработчик для консоли
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setLevel(logging.INFO)
    console_handler.setFormatter(formatter)

    # Обработчик для файла
            file_handler   =   logging.FileHandler(f'app_{datetime.now().
strftime("%Y%m%d")}.log')
    file_handler.setLevel(logging.DEBUG)
    file_handler.setFormatter(formatter)

    # Добавляем обработчики к логгеру
    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

    return logger
class DatabaseConnection:
    """Пример класса с настроенным логированием"""

    def __init__(self, host, port, database):
        self.host = host
        self.port = port
        self.database = database
        self.connected = False
```

```
        # Получаем логгер для этого класса
           self.logger = logging.getLogger(f'{__name__}.{self.__class__.__
name__}')

    def connect(self):
        """Имитация подключения к базе данных"""
       self.logger.info(f"Attempting to connect to {self.host}:{self.port}/
{self.database}")

        try:
            # Имитируем различные сценарии
            if self.port < 1024:
                 self.logger.warning("Using privileged port, ensure proper
permissions")

            if self.host == 'localhost':
                self.logger.debug("Connecting to local database")
            else:
                self.logger.debug(f"Connecting to remote database at {self.
host}")

            # Имитируем успешное подключение
            time.sleep(0.1)
            self.connected = True
           self.logger.info("Database connection established successfully")

        except Exception as e:
            self.logger.error(f"Failed to connect to database: {e}")
            raise

    def execute_query(self, query):
        """Выполнение запроса с логированием"""
        if not self.connected:
         self.logger.error("Attempting to execute query without connection")
            raise RuntimeError("Not connected to database")

        self.logger.debug(f"Executing query: {query[:50]}...")

        # Имитируем выполнение запроса
        if 'DROP' in query.upper():
            self.logger.critical(f"DANGEROUS QUERY DETECTED: {query}")
        elif 'DELETE' in query.upper():
            self.logger.warning(f"Destructive query: {query}")

        self.logger.info("Query executed successfully")
        return f"Results for: {query[:30]}..."

    def close(self):
        """Закрытие соединения"""
        if self.connected:
            self.logger.info("Closing database connection")
```

```
            self.connected = False
        else:
            self.logger.warning("Attempted to close non-existent connection")
# Настраиваем логирование
logger = setup_advanced_logging()
# Демонстрация использования
logger.info("Application starting...")
db = DatabaseConnection('localhost', 5432, 'myapp')
try:
    db.connect()

    # Выполняем различные запросы
    db.execute_query('SELECT * FROM users WHERE active = 1')
    db.execute_query('DELETE FROM temp_data WHERE created < "2023-01-01"')
    db.execute_query('DROP TABLE old_logs')  # Опасный запрос

finally:
    db.close()
    logger.info("Application finished")
```

Фильтры и обработчики

Система логирования позволяет создавать сложные конфигурации с различными обработчиками и фильтрами:

```
import logging
import json
from datetime import datetime
class JSONFormatter(logging.Formatter):
    """Кастомный форматтер для JSON-вывода"""

    def format(self, record):
        """Форматирует запись в JSON"""
        log_entry = {
            'timestamp': datetime.fromtimestamp(record.created).isoformat(),
            'level': record.levelname,
            'logger': record.name,
            'message': record.getMessage(),
            'module': record.module,
            'function': record.funcName,
            'line': record.lineno
        }

        # Добавляем информацию об исключении, если есть
        if record.exc_info:
            log_entry['exception'] = self.formatException(record.exc_info)

        return json.dumps(log_entry)
class SensitiveDataFilter(logging.Filter):
```

```
    """Фильтр для удаления чувствительных данных"""
    def __init__(self):
        super().__init__()
        self.sensitive_patterns = ['password', 'token', 'secret', 'key']

    def filter(self, record):
        """Фильтрует чувствительные данные"""
        message = record.getMessage().lower()

        # Проверяем на наличие чувствительных данных
        for pattern in self.sensitive_patterns:
            if pattern in message:
                record.msg = record.msg.replace(
                    record.msg[record.msg.lower().find(pattern):],
                    f"{pattern.upper()}_REDACTED"
                )

        return True
class UserAuthenticationSystem:
    """Система аутентификации с продвинутым логированием"""

    def __init__(self):
        # Настройка логгера
        self.logger = logging.getLogger('AuthSystem')
        self.logger.setLevel(logging.DEBUG)

        # JSON-форматтер для структурированных логов
        json_formatter = JSONFormatter()

        # Обработчик для файла с JSON-логами
        json_handler = logging.FileHandler('auth_logs.json')
        json_handler.setFormatter(json_formatter)
        json_handler.setLevel(logging.INFO)

        # Консольный обработчик
        console_handler = logging.StreamHandler()
        console_formatter = logging.Formatter(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )
        console_handler.setFormatter(console_formatter)
        console_handler.setLevel(logging.WARNING)

        # Добавляем фильтр для чувствительных данных
        sensitive_filter = SensitiveDataFilter()
        json_handler.addFilter(sensitive_filter)
        console_handler.addFilter(sensitive_filter)

        # Добавляем обработчики
        self.logger.addHandler(json_handler)
        self.logger.addHandler(console_handler)

        # Имитация базы пользователей
```

```
        self.users_db = {
            'alice': {'password': 'secret123', 'role': 'admin'},
            'bob': {'password': 'mypassword', 'role': 'user'}
        }

    def authenticate_user(self, username, password):
        """Аутентифицирует пользователя"""
        self.logger.info(f"Authentication attempt for user: {username}")

        if username not in self.users_db:
                self.logger.warning(f"Authentication failed: unknown user
{username}")
            return False

        stored_password = self.users_db[username]['password']

        if password != stored_password:
            self.logger.warning(
               f"Authentication failed for {username}: password mismatch. "
                f"Provided password: {password}"  # Будет отфильтровано
            )
            return False

        user_role = self.users_db[username]['role']
         self.logger.info(f"Authentication successful for {username} (role:
{user_role})")

        return True

    def generate_token(self, username):
        """Генерирует токен для пользователя"""
        if username in self.users_db:
            token = f"token_{username}_{datetime.now().timestamp()}"
             self.logger.debug(f"Generated token for {username}: {token}")
# Будет отфильтровано
            return token
        return None
# Демонстрация работы системы
auth_system = UserAuthenticationSystem()
# Тестируем различные сценарии
test_scenarios = [
    ('alice', 'secret123'),     # Успешная аутентификация
    ('bob', 'wrongpassword'),   # Неверный пароль
    ('charlie', 'anypassword'), # Несуществующий пользователь
    ('alice', 'secret123')      # Повторная успешная аутентификация
]
for username, password in test_scenarios:
    success = auth_system.authenticate_user(username, password)

    if success:
        token = auth_system.generate_token(username)
```

```
        print(f"Login successful for {username}")
    else:
        print(f"Login failed for {username}")
```

Логирование — это инвестиция в будущее вашего кода. Хорошо настроенная система логирования сэкономит вам часы отладки и поможет понять поведение программы в продакшене. В следующем разделе мы изучим автоматическое тестирование — еще один важный инструмент обеспечения качества кода.

10.4. Юнит-тестирование: модуль unittest

Тестирование — это процесс проверки программы на соответствие ожиданиям. **Юнит-тестирование (unit testing)** — это тестирование отдельных компонентов программы изолированно от других частей. Каждый юнит-тест проверяет один небольшой аспект функциональности, обычно отдельную функцию или метод класса.

Представьте себе сборочную линию автомобильного завода. Каждая деталь проверяется отдельно перед установкой в автомобиль — тормозная система, двигатель, электроника. Только после того, как каждая деталь прошла индивидуальную проверку, они объединяются в готовый продукт. Юнит-тестирование работает по тому же принципу: мы проверяем каждую функцию отдельно, чтобы убедиться в корректности всей программы.

Философия тестирования

Тестирование служит нескольким важным целям. Оно обеспечивает уверенность в том, что код работает правильно, создает документацию поведения кода через примеры использования, облегчает рефакторинг — изменения кода без страха что-то сломать — и помогает находить ошибки на ранних стадиях разработки.

Хорошие тесты должны быть быстрыми — выполняться за секунды, независимыми — не зависеть друг от друга, повторяемыми — давать одинаковые результаты при многократном запуске и понятными — четко показывать, что именно тестируется.

Модуль unittest: основы

Python предоставляет встроенный модуль `unittest`, который реализует фреймворк тестирования, основанный на концепциях `xUnit`. Начнем с простого примера:

```
import unittest
def add_numbers(a, b):
    """Складывает два числа"""
    return a + b
def divide_numbers(a, b):
    """Делит два числа"""
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
class TestMathOperations(unittest.TestCase):
    """Тесты для математических операций"""
    def test_add_positive_numbers(self):
```

```
        """Тест сложения положительных чисел"""
        result = add_numbers(3, 5)
        self.assertEqual(result, 8)

    def test_add_negative_numbers(self):
        """Тест сложения отрицательных чисел"""
        result = add_numbers(-3, -5)
        self.assertEqual(result, -8)

    def test_add_mixed_numbers(self):
        """Тест сложения положительного и отрицательного числа"""
        result = add_numbers(10, -3)
        self.assertEqual(result, 7)

    def test_divide_normal_case(self):
        """Тест обычного деления"""
        result = divide_numbers(10, 2)
        self.assertEqual(result, 5.0)

    def test_divide_by_zero(self):
        """Тест деления на ноль"""
        with self.assertRaises(ValueError) as context:
            divide_numbers(10, 0)

        self.assertEqual(str(context.exception), "Cannot divide by zero")
# Запуск тестов
if __name__ == '__main__':
    unittest.main()
```

Структура тестов

Каждый тест-класс наследуется от `unittest.TestCase` и содержит методы, имена которых начинаются с `test_`. Каждый тестовый метод проверяет одну конкретную функциональность.

Жизненный цикл теста

Рассмотрим более сложный пример с настройкой и очисткой:

```
import unittest
import tempfile
import os
class BankAccount:
    """Класс банковского счета для демонстрации тестирования"""

    def __init__(self, owner, initial_balance=0):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative")

        self.owner = owner
        self.balance = initial_balance
        self.transaction_history = []
```

```
        self._add_transaction("Account opened", initial_balance)

    def deposit(self, amount):
        """Внесение денег на счет"""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")

        self.balance += amount
        self._add_transaction("Deposit", amount)
        return self.balance

    def withdraw(self, amount):
        """Снятие денег со счета"""
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")

        if amount > self.balance:
            raise ValueError("Insufficient funds")

        self.balance -= amount
        self._add_transaction("Withdrawal", -amount)
        return self.balance

    def get_balance(self):
        """Получение текущего баланса"""
        return self.balance

    def get_transaction_history(self):
        """Получение истории транзакций"""
        return self.transaction_history.copy()

    def _add_transaction(self, transaction_type, amount):
        """Добавляет транзакцию в историю"""
        from datetime import datetime
        self.transaction_history.append({
            'type': transaction_type,
            'amount': amount,
            'timestamp': datetime.now(),
            'balance_after': self.balance
        })
class TestBankAccount(unittest.TestCase):
    """Тесты для класса BankAccount"""

    def setUp(self):
        """Настройка перед каждым тестом"""
        # Этот метод вызывается перед каждым тестом
        self.account = BankAccount("Test User", 100)
        self.empty_account = BankAccount("Empty User", 0)

    def tearDown(self):
        """Очистка после каждого теста"""
```

```
        # Этот метод вызывается после каждого теста
        # В данном случае особой очистки не требуется,
        # но здесь можно закрывать файлы, соединения с БД и т.д.
        pass

    def test_account_creation_valid(self):
        """Тест создания счета с валидными данными"""
        account = BankAccount("John Doe", 500)

        self.assertEqual(account.owner, "John Doe")
        self.assertEqual(account.get_balance(), 500)
        self.assertEqual(len(account.get_transaction_history()), 1)

    def test_account_creation_negative_balance(self):
        """Тест создания счета с отрицательным балансом"""
        with self.assertRaises(ValueError) as context:
            BankAccount("John Doe", -100)

        self.assertIn("negative", str(context.exception).lower())

    def test_deposit_valid_amount(self):
        """Тест внесения валидной суммы"""
        initial_balance = self.account.get_balance()
        new_balance = self.account.deposit(50)

        self.assertEqual(new_balance, initial_balance + 50)
        self.assertEqual(self.account.get_balance(), initial_balance + 50)

    def test_deposit_zero_amount(self):
        """Тест внесения нулевой суммы"""
        with self.assertRaises(ValueError):
            self.account.deposit(0)

    def test_deposit_negative_amount(self):
        """Тест внесения отрицательной суммы"""
        with self.assertRaises(ValueError):
            self.account.deposit(-10)

    def test_withdraw_valid_amount(self):
        """Тест снятия валидной суммы"""
        initial_balance = self.account.get_balance()
        new_balance = self.account.withdraw(30)

        self.assertEqual(new_balance, initial_balance - 30)
        self.assertEqual(self.account.get_balance(), initial_balance - 30)

    def test_withdraw_insufficient_funds(self):
        """Тест снятия суммы больше баланса"""
        with self.assertRaises(ValueError) as context:
```

```
            self.account.withdraw(200)  # больше чем 100
        self.assertIn("insufficient", str(context.exception).lower())

    def test_withdraw_from_empty_account(self):
        """Тест снятия с пустого счета"""
        with self.assertRaises(ValueError):
            self.empty_account.withdraw(1)

    def test_transaction_history(self):
        """Тест истории транзакций"""
        # Выполняем несколько операций
        self.account.deposit(25)
        self.account.withdraw(10)
        self.account.deposit(5)

        history = self.account.get_transaction_history()

        # Проверяем количество транзакций (включая открытие счета)
        self.assertEqual(len(history), 4)

        # Проверяем типы транзакций
            expected_types = ["Account opened", "Deposit", "Withdrawal",
"Deposit"]
        actual_types = [t['type'] for t in history]
        self.assertEqual(actual_types, expected_types)

        # Проверяем финальный баланс
        self.assertEqual(history[-1]['balance_after'], 120)
```

Методы утверждений (Assertions)

Модуль `unittest` предоставляет множество методов для проверки различных условий:

```
import unittest
import math
from collections import Counter
def analyze_text(text):
    """Анализирует текст и возвращает статистику"""
    if not isinstance(text, str):
        raise TypeError("Input must be a string")

    words = text.lower().split()

    return {
        'word_count': len(words),
        'character_count': len(text),
        'unique_words': len(set(words)),
        'word_frequency': dict(Counter(words)),
       'average_word_length': sum(len(word) for word in words) / len(words)
if words else 0,
```

```
        'has_digits': any(char.isdigit() for char in text),
        'is_empty': len(text.strip()) == 0
    }
def calculate_circle_properties(radius):
    """Вычисляет свойства окружности"""
    if radius < 0:
        raise ValueError("Radius cannot be negative")

    return {
        'area': math.pi * radius ** 2,
        'circumference': 2 * math.pi * radius,
        'diameter': 2 * radius
    }
class TestAdvancedAssertions(unittest.TestCase):
    """Демонстрация различных методов утверждений"""

    def test_text_analysis_basic(self):
        """Тест базового анализа текста"""
        result = analyze_text("Hello world hello")

        # assertEqual - точное равенство
        self.assertEqual(result['word_count'], 3)
        self.assertEqual(result['unique_words'], 2)

        # assertIn - проверка содержания
        self.assertIn('hello', result['word_frequency'])
        self.assertIn('world', result['word_frequency'])

        # assertTrue/assertFalse - булевы значения
        self.assertFalse(result['is_empty'])
        self.assertFalse(result['has_digits'])

    def test_text_analysis_with_numbers(self):
        """Тест анализа текста с числами"""
        result = analyze_text("Test 123 data")

        self.assertTrue(result['has_digits'])
        self.assertEqual(result['word_count'], 3)

    def test_text_analysis_empty_string(self):
        """Тест анализа пустой строки"""
        result = analyze_text("")

        self.assertTrue(result['is_empty'])
        self.assertEqual(result['word_count'], 0)
        self.assertEqual(result['average_word_length'], 0)

    def test_text_analysis_invalid_input(self):
        """Тест анализа с невалидным вводом"""
        with self.assertRaises(TypeError):
            analyze_text(123)
```

```
        with self.assertRaises(TypeError):
            analyze_text(None)

    def test_circle_properties_valid_radius(self):
        """Тест вычисления свойств окружности"""
        result = calculate_circle_properties(5)

        # assertAlmostEqual - для чисел с плавающей точкой
        self.assertAlmostEqual(result['area'], 78.53981633974483, places=5)
         self.assertAlmostEqual(result['circumference'], 31.41592653589793,
places=5)

        # assertEqual для точных значений
        self.assertEqual(result['diameter'], 10)

    def test_circle_properties_zero_radius(self):
        """Тест с нулевым радиусом"""
        result = calculate_circle_properties(0)

        self.assertEqual(result['area'], 0)
        self.assertEqual(result['circumference'], 0)
        self.assertEqual(result['diameter'], 0)

    def test_circle_properties_negative_radius(self):
        """Тест с отрицательным радиусом"""
        with self.assertRaises(ValueError):
            calculate_circle_properties(-5)

    def test_collections_assertions(self):
        """Тест утверждений для коллекций"""
        list1 = [1, 2, 3, 4, 5]
        list2 = [5, 4, 3, 2, 1]
        dict1 = {'a': 1, 'b': 2, 'c': 3}

        # assertListEqual - для списков
        self.assertListEqual(sorted(list1), sorted(list2))

        # assertDictEqual - для словарей
        expected_dict = {'a': 1, 'b': 2, 'c': 3}
        self.assertDictEqual(dict1, expected_dict)

        # assertCountEqual - проверка содержимого без учета порядка
        self.assertCountEqual(list1, list2)

        # assertGreater/assertLess - сравнения
        self.assertGreater(len(list1), 3)
        self.assertLess(dict1['a'], dict1['b'])

        # assertIsNone/assertIsNotNone
        none_value = None
        not_none_value = "something"
```

```
        self.assertIsNone(none_value)
        self.assertIsNotNone(not_none_value)
```

Пропуск тестов и условное выполнение

Иногда тесты нужно пропускать при определенных условиях:

```
import unittest
import sys
import os
class FileManager:
    """Класс для управления файлами"""

    def __init__(self, base_path):
        self.base_path = base_path

    def create_file(self, filename, content=""):
        """Создает файл с содержимым"""
        full_path = os.path.join(self.base_path, filename)
        with open(full_path, 'w', encoding='utf-8') as f:
            f.write(content)
        return full_path

    def read_file(self, filename):
        """Читает содержимое файла"""
        full_path = os.path.join(self.base_path, filename)
        with open(full_path, 'r', encoding='utf-8') as f:
            return f.read()

    def delete_file(self, filename):
        """Удаляет файл"""
        full_path = os.path.join(self.base_path, filename)
        if os.path.exists(full_path):
            os.remove(full_path)
            return True
        return False

    def list_files(self):
        """Возвращает список файлов в директории"""
        if os.path.exists(self.base_path):
            return os.listdir(self.base_path)
        return []
class TestFileManager(unittest.TestCase):
    """Тесты для FileManager с примерами пропуска тестов"""

    def setUp(self):
        """Настройка тестовой среды"""
        import tempfile
        self.test_dir = tempfile.mkdtemp()
        self.file_manager = FileManager(self.test_dir)
```

```
    def tearDown(self):
        """Очистка после тестов"""
        import shutil
        shutil.rmtree(self.test_dir, ignore_errors=True)

    def test_create_and_read_file(self):
        """Тест создания и чтения файла"""
        content = "Test content for file"
        filename = "test_file.txt"

        # Создаем файл
        created_path = self.file_manager.create_file(filename, content)
        self.assertTrue(os.path.exists(created_path))

        # Читаем файл
        read_content = self.file_manager.read_file(filename)
        self.assertEqual(read_content, content)

    def test_delete_file(self):
        """Тест удаления файла"""
        filename = "file_to_delete.txt"

        # Создаем файл
        self.file_manager.create_file(filename, "content")

        # Проверяем, что файл существует
        file_path = os.path.join(self.test_dir, filename)
        self.assertTrue(os.path.exists(file_path))

        # Удаляем файл
        deleted = self.file_manager.delete_file(filename)
        self.assertTrue(deleted)
        self.assertFalse(os.path.exists(file_path))

    @unittest.skipIf(sys.platform.startswith("win"), "Test not supported on
Windows")
    def test_unix_specific_functionality(self):
        """Тест функциональности, специфичной для Unix-систем"""
        # Этот тест будет пропущен на Windows
        filename = "unix_test_file"
        self.file_manager.create_file(filename, "Unix content")

        file_path = os.path.join(self.test_dir, filename)
        # Проверяем Unix-специфичные права доступа
        file_mode = oct(os.stat(file_path).st_mode)
        self.assertIn('644', file_mode)  # Типичные права для файла

    @unittest.skipUnless(os.name == 'posix', "Requires POSIX system")
    def test_posix_file_permissions(self):
        """Тест прав доступа к файлам в POSIX-системах"""
        filename = "permission_test.txt"
```

```
        file_path = self.file_manager.create_file(filename, "test")

        # Изменяем права доступа
        os.chmod(file_path, 0o755)

        # Проверяем права
        file_stat = os.stat(file_path)
        self.assertTrue(file_stat.st_mode & 0o755)

    @unittest.skip("Feature not yet implemented")
    def test_future_functionality(self):
        """Тест функциональности, которая еще не реализована"""
        # Этот тест всегда будет пропущен
        pass

    def test_conditional_skip_inside_test(self):
        """Демонстрация условного пропуска внутри теста"""
        if not hasattr(os, 'symlink'):
            self.skipTest("Symbolic links not supported on this platform")

        # Код теста для символических ссылок
        filename = "original_file.txt"
        link_name = "link_to_file.txt"

            file_path = self.file_manager.create_file(filename, "original
content")
        link_path = os.path.join(self.test_dir, link_name)

        # Создаем символическую ссылку
        os.symlink(file_path, link_path)
        self.assertTrue(os.path.islink(link_path))
```

Группировка тестов и тестовые наборы

Для организации больших тестовых наборов используются `TestSuite`:

```
import unittest
from io import StringIO
class CalculatorEngine:
    """Простой калькулятор для демонстрации группировки тестов"""

    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def subtract(a, b):
        return a - b

    @staticmethod
    def multiply(a, b):
```

```
        return a * b

    @staticmethod
    def divide(a, b):
        if b == 0:
            raise ZeroDivisionError("Division by zero")
        return a / b

    @staticmethod
    def power(base, exponent):
        return base ** exponent

    @staticmethod
    def square_root(number):
        if number < 0:
            raise ValueError("Cannot calculate square root of negative
number")
        return number ** 0.5
class TestBasicOperations(unittest.TestCase):
    """Тесты базовых арифметических операций"""

    def test_addition(self):
        self.assertEqual(CalculatorEngine.add(5, 3), 8)
        self.assertEqual(CalculatorEngine.add(-1, 1), 0)
        self.assertEqual(CalculatorEngine.add(0, 0), 0)

    def test_subtraction(self):
        self.assertEqual(CalculatorEngine.subtract(5, 3), 2)
        self.assertEqual(CalculatorEngine.subtract(-1, 1), -2)
        self.assertEqual(CalculatorEngine.subtract(0, 0), 0)

    def test_multiplication(self):
        self.assertEqual(CalculatorEngine.multiply(4, 3), 12)
        self.assertEqual(CalculatorEngine.multiply(-2, 3), -6)
        self.assertEqual(CalculatorEngine.multiply(0, 5), 0)

    def test_division(self):
        self.assertEqual(CalculatorEngine.divide(15, 3), 5)
        self.assertEqual(CalculatorEngine.divide(-6, 2), -3)

        with self.assertRaises(ZeroDivisionError):
            CalculatorEngine.divide(10, 0)
class TestAdvancedOperations(unittest.TestCase):
    """Тесты продвинутых математических операций"""

    def test_power(self):
        self.assertEqual(CalculatorEngine.power(2, 3), 8)
        self.assertEqual(CalculatorEngine.power(5, 0), 1)
        self.assertEqual(CalculatorEngine.power(10, 2), 100)

    def test_square_root(self):
        self.assertAlmostEqual(CalculatorEngine.square_root(9), 3.0)
```

```
        self.assertAlmostEqual(CalculatorEngine.square_root(2),
1.4142135623730951)
        self.assertEqual(CalculatorEngine.square_root(0), 0)

        with self.assertRaises(ValueError):
            CalculatorEngine.square_root(-4)
def create_test_suite():
    """Создает кастомный набор тестов"""
    suite = unittest.TestSuite()

    # Добавляем конкретные тесты
    suite.addTest(TestBasicOperations('test_addition'))
    suite.addTest(TestBasicOperations('test_multiplication'))
    suite.addTest(TestAdvancedOperations('test_power'))

    return suite
def run_specific_tests():
    """Запускает определенные тесты с кастомным выводом"""
    # Создаем тестовый набор
    suite = create_test_suite()

    # Создаем кастомный runner с подробным выводом
    stream = StringIO()
    runner = unittest.TextTestRunner(stream=stream, verbosity=2)

    # Запускаем тесты
    result = runner.run(suite)

    # Выводим результаты
    print("=== Custom Test Run Results ===")
    print(stream.getvalue())
    print(f"Tests run: {result.testsRun}")
    print(f"Failures: {len(result.failures)}")
    print(f"Errors: {len(result.errors)}")

    return result.wasSuccessful()
# Демонстрация различных способов запуска тестов
if __name__ == '__main__':
    print("Running custom test suite...")
    success = run_specific_tests()

    print("\n" + "="*50)
    print("Running all tests...")

    # Запускаем все тесты
    unittest.main(argv=[''], exit=False, verbosity=2)
```

Модуль `unittest` предоставляет мощную и гибкую платформу для написания тестов. Он следует проверенным принципам тестирования и интегрируется с большинством IDE и систем непрерывной интеграции. Хотя синтаксис может показаться

многословным по сравнению с более современными фреймворками, unittest остается надежным выбором для серьезных проектов.

Ключевые принципы эффективного юнит-тестирования включают написание тестов для каждой публичной функции или метода, проверку как нормальных случаев, так и граничных условий и ошибок, поддержание независимости тестов друг от друга и использование понятных имен тестов, которые объясняют, что именно проверяется.

В следующем разделе мы изучим более современный фреймворк тестирования pytest, который предлагает более простой синтаксис и расширенные возможности, оставаясь при этом совместимым с тестами unittest.

10.5. Pytest: современный фреймворк тестирования

Pytest — это современный и мощный фреймворк тестирования для Python, который значительно упрощает написание и выполнение тестов. В отличие от unittest, pytest использует более простой синтаксис, не требует создания классов для всех тестов и предоставляет богатую экосистему плагинов.

Основные преимущества pytest включают простой синтаксис без избыточного кода, автоматическое обнаружение тестов, информативные сообщения об ошибках, мощную систему фикстур для настройки тестов и обширную библиотеку плагинов.

Основы pytest

Установка pytest выполняется через pip:

```
pip install pytest
Простейший тест в pytest выглядит как обычная функция:
# test_basic.py
def add_numbers(a, b):
    """Складывает два числа"""
    return a + b
def test_add_positive_numbers():
    """Тест сложения положительных чисел"""
    result = add_numbers(3, 5)
    assert result == 8
def test_add_negative_numbers():
    """Тест сложения отрицательных чисел"""
    result = add_numbers(-3, -5)
    assert result == -8
def test_add_zero():
    """Тест сложения с нулем"""
    assert add_numbers(5, 0) == 5
    assert add_numbers(0, 0) == 0
```

Запуск тестов:

```
pytest test_basic.py
```

Организация тестов

```
# calculator.py
class Calculator:
    """Простой калькулятор"""

    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
# test_calculator.py
import pytest
from calculator import Calculator
class TestCalculator:
    """Тесты для класса Calculator"""

    def setup_method(self):
        """Настройка перед каждым тестом"""
        self.calc = Calculator()

    def test_addition(self):
        """Тест сложения"""
        assert self.calc.add(2, 3) == 5
        assert self.calc.add(-1, 1) == 0

    def test_subtraction(self):
        """Тест вычитания"""
        assert self.calc.subtract(5, 3) == 2
        assert self.calc.subtract(0, 5) == -5

    def test_multiplication(self):
        """Тест умножения"""
        assert self.calc.multiply(4, 3) == 12
        assert self.calc.multiply(-2, 3) == -6

    def test_division(self):
        """Тест деления"""
        assert self.calc.divide(10, 2) == 5
        assert self.calc.divide(7, 2) == 3.5

    def test_division_by_zero(self):
        """Тест деления на ноль"""
        with pytest.raises(ValueError, match="Cannot divide by zero"):
            self.calc.divide(10, 0)
```

Фикстуры (Fixtures)

Фикстуры — это механизм `pytest` для подготовки данных и ресурсов, необходимых для тестов. Они заменяют методы `setUp/tearDown` из `unittest` и предоставляют более гибкие возможности.

Базовые фикстуры

```
import pytest
import tempfile
import os
from datetime import datetime
@pytest.fixture
def sample_data():
    """Фикстура с тестовыми данными"""
    return {
        'users': [
            {'id': 1, 'name': 'Alice', 'email': 'alice@example.com'},
            {'id': 2, 'name': 'Bob', 'email': 'bob@example.com'},
            {'id': 3, 'name': 'Charlie', 'email': 'charlie@example.com'}
        ],
        'products': [
            {'id': 101, 'name': 'Laptop', 'price': 999.99},
            {'id': 102, 'name': 'Mouse', 'price': 29.99}
        ]
    }
@pytest.fixture
def temp_file():
    """Фикстура для временного файла"""
    # Создаем временный файл
    fd, path = tempfile.mkstemp(suffix='.txt')

    with os.fdopen(fd, 'w') as f:
        f.write("Test content\nSecond line\nThird line")

    yield path  # Возвращаем путь к файлу

    # Очистка после теста
    if os.path.exists(path):
        os.remove(path)
class DataProcessor:
    """Класс для обработки данных"""

    def __init__(self, data):
        self.data = data

    def get_user_by_id(self, user_id):
        """Получить пользователя по ID"""
        users = self.data.get('users', [])
        return next((u for u in users if u['id'] == user_id), None)

    def get_user_emails(self):
```

```
        """Получить список email-пользователей"""
        users = self.data.get('users', [])
        return [user['email'] for user in users]

    def get_total_product_value(self):
        """Получить общую стоимость продуктов"""
        products = self.data.get('products', [])
        return sum(product['price'] for product in products)
def test_get_user_by_id(sample_data):
    """Тест получения пользователя по ID"""
    processor = DataProcessor(sample_data)

    user = processor.get_user_by_id(1)
    assert user is not None
    assert user['name'] == 'Alice'
    assert user['email'] == 'alice@example.com'

    # Тест несуществующего пользователя
    assert processor.get_user_by_id(999) is None
def test_get_user_emails(sample_data):
    """Тест получения списка email"""
    processor = DataProcessor(sample_data)

    emails = processor.get_user_emails()
     expected = ['alice@example.com', 'bob@example.com', 'charlie@example.
com']
    assert emails == expected
def test_file_reading(temp_file):
    """Тест чтения файла"""
    with open(temp_file, 'r') as f:
        content = f.read()

    lines = content.split('\n')
    assert len(lines) == 3
    assert lines[0] == "Test content"
    assert lines[1] == "Second line"
```

Фикстуры с областями видимости

```
@pytest.fixture(scope="session")
def database_connection():
    """Соединение с БД на уровне сессии"""
    print("\nConnecting to database...")
    # Имитация подключения к БД
    connection = {"status": "connected", "db": "test_db"}
    yield connection
    print("\nClosing database connection...")
@pytest.fixture(scope="module")
def api_client():
    """API-клиент на уровне модуля"""
    print("\nInitializing API client...")
```

```
    client = {"base_url": "https://api.test.com", "token": "test_token"}
    yield client
    print("\nShutting down API client...")
@pytest.fixture
def current_timestamp():
    """Текущая временная метка для каждого теста"""
    return datetime.now().timestamp()
def test_database_query(database_connection, current_timestamp):
    """Тест запроса к БД"""
    assert database_connection["status"] == "connected"
    assert current_timestamp > 0
def test_api_request(api_client, current_timestamp):
    """Тест API запроса"""
    assert api_client["base_url"].startswith("https://")
    assert current_timestamp > 0
def test_another_database_query(database_connection):
    """Еще один тест БД"""
    assert database_connection["db"] == "test_db"
```

Параметризация тестов

Параметризация позволяет запускать один тест с разными наборами данных:

```
import pytest
import math
def validate_email(email):
    """Простая валидация email"""
    if not email or '@' not in email:
        return False

    parts = email.split('@')
    if len(parts) != 2:
        return False

    username, domain = parts
    if not username or not domain:
        return False

    if '.' not in domain:
        return False

    return True
def calculate_discount(price, discount_percent):
    """Вычисляет цену со скидкой"""
    if price < 0:
        raise ValueError("Price cannot be negative")
    if not (0 <= discount_percent <= 100):
        raise ValueError("Discount must be between 0 and 100")

    return price * (1 - discount_percent / 100)
@pytest.mark.parametrize("email,expected", [
```

```
    ("test@example.com", True),
    ("user@domain.org", True),
    ("invalid.email", False),
    ("", False),
    ("@domain.com", False),
    ("user@", False),
    ("user@@domain.com", False),
    ("user@domain", False),
])
def test_email_validation(email, expected):
    """Параметризованный тест валидации email"""
    assert validate_email(email) == expected
@pytest.mark.parametrize("price,discount,expected", [
    (100, 10, 90),    # 10% скидка
    (100, 0, 100),    # Без скидки
    (100, 100, 0),    # Полная скидка
    (50, 20, 40),     # 20% скидка
    (0, 50, 0),       # Нулевая цена
])
def test_discount_calculation(price, discount, expected):
    """Параметризованный тест расчета скидки"""
    result = calculate_discount(price, discount)
    assert abs(result - expected) < 0.01  # Учитываем погрешность float
@pytest.mark.parametrize("price,discount", [
    (-100, 10),    # Отрицательная цена
    (100, -10),    # Отрицательная скидка
    (100, 150),    # Скидка больше 100%
])
def test_discount_calculation_errors(price, discount):
    """Параметризованный тест ошибок расчета скидки"""
    with pytest.raises(ValueError):
        calculate_discount(price, discount)
```

Комбинированная параметризация

```
@pytest.mark.parametrize("base", [2, 3, 5])
@pytest.mark.parametrize("exponent", [0, 1, 2, 3])
def test_power_function(base, exponent):
    """Тест функции возведения в степень с комбинированными параметрами"""
    result = base ** exponent

    if exponent == 0:
        assert result == 1
    elif exponent == 1:
        assert result == base
    else:
       # Проверяем, что результат больше базы при положительной степени > 1
        if base > 1:
            assert result > base
class TestStringOperations:
    """Тесты строковых операций с параметризацией"""
```

```
    @pytest.mark.parametrize("text,expected_length", [
        ("hello", 5),
        ("", 0),
        ("Python programming", 18),
        ("Тест на русском", 14),
    ])
    def test_string_length(self, text, expected_length):
        """Тест длины строки"""
        assert len(text) == expected_length

    @pytest.mark.parametrize("text,char,expected_count", [
        ("hello world", "l", 3),
        ("python", "p", 1),
        ("programming", "m", 2),
        ("test", "x", 0),
    ])
    def test_character_count(self, text, char, expected_count):
        """Тест подсчета символов в строке"""
        assert text.count(char) == expected_count
```

Маркеры (Markers) и плагины

Pytest позволяет маркировать тесты для группировки и условного выполнения:

```
import pytest
import sys
import os
@pytest.mark.slow
def test_slow_operation():
    """Медленный тест"""
    import time
    time.sleep(0.1)  # Имитация медленной операции
    assert True
@pytest.mark.integration
def test_database_integration():
    """Интеграционный тест"""
    # Имитация теста интеграции с БД
    assert True
@pytest.mark.unit
def test_unit_function():
    """Юнит-тест"""
    def simple_function(x):
        return x * 2

    assert simple_function(5) == 10
@pytest.mark.skipif(sys.platform == "win32", reason="Not supported on
Windows")
def test_unix_specific():
    """Тест только для Unix-систем"""
    assert os.name == "posix"
```

```
@pytest.mark.xfail(reason="Known bug in external library")
def test_expected_to_fail():
    """Тест, который ожидаемо падает"""
    assert 1 == 2  # Намеренно неверное утверждение
class TestWithCustomMarkers:
    """Класс с кастомными маркерами"""

    @pytest.mark.api
    @pytest.mark.slow
    def test_api_endpoint(self):
        """Тест API эндпоинта"""
        # Имитация HTTP-запроса
        response = {"status": 200, "data": "success"}
        assert response["status"] == 200

    @pytest.mark.unit
    @pytest.mark.fast
    def test_quick_calculation(self):
        """Быстрый вычислительный тест"""
        assert 2 + 2 == 4
# Запуск определенных маркеров:
# pytest -m "unit"           # Только unit-тесты
# pytest -m "slow"           # Только медленные тесты
# pytest -m "not slow"       # Все, кроме медленных
# pytest -m "unit and fast"  # Unit-тесты, которые также быстрые
```

Конфигурация pytest

```
# conftest.py - файл конфигурации pytest
import pytest
@pytest.fixture(autouse=True)
def setup_test_environment():
    """Автоматически выполняется для каждого теста"""
    print("\nSetting up test environment...")
    yield
    print("\nCleaning up after test...")
@pytest.fixture
def shared_data():
    """Общие данные для тестов"""
    return {
        "test_mode": True,
        "debug": False,
        "version": "1.0.0"
    }
def pytest_configure(config):
    """Настройка pytest"""
    config.addinivalue_line(
        "markers", "slow: marks tests as slow"
    )
    config.addinivalue_line(
        "markers", "integration: marks tests as integration tests"
    )
```

```
    config.addinivalue_line(
        "markers", "api: marks tests as API tests"
    )
# pytest.ini - файл конфигурации
"""
[tool:pytest]
testpaths = tests
python_files = test_*.py *_test.py
python_classes = Test*
python_functions = test_*
addopts = -v --tb=short
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    integration: marks tests as integration tests
    unit: marks tests as unit tests
    api: marks tests as API tests
"""
```

Pytest предоставляет элегантный и мощный способ написания тестов. Его простой синтаксис, гибкая система фикстур и богатая экосистема плагинов делают его предпочтительным выбором для многих Python-проектов. Ключевые преимущества включают минимальный синтаксис без избыточного кода, автоматическое обнаружение тестов, информативные сообщения об ошибках и мощную систему параметризации для тестирования различных сценариев.

10.6. Покрытие кода тестами

Покрытие кода тестами (code coverage) — это метрика, показывающая, какая часть кода выполняется во время тестирования. Это важный инструмент для оценки качества тестов и поиска неопротестированных участков кода.

Покрытие помогает выявить неопротестированный код, повысить уверенность в качестве тестов, найти мертвый или недостижимый код и контролировать качество тестирования в проекте.

Установка и использование coverage.py

Coverage.py — это стандартный инструмент для измерения покрытия кода в Python:

```
pip install coverage
```

Базовое использование

```
# math_operations.py
def add(a, b):
    """Сложение двух чисел"""
    return a + b
def subtract(a, b):
    """Вычитание двух чисел"""
    return a - b
def multiply(a, b):
```

```
    """Умножение двух чисел"""
    return a * b
def divide(a, b):
    """Деление двух чисел"""
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
def factorial(n):
    """Вычисление факториала"""
    if n < 0:
        raise ValueError("Factorial not defined for negative numbers")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
def is_prime(n):
    """Проверка числа на простоту"""
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    for i in range(3, int(n ** 0.5) + 1, 2):
        if n % i == 0:
            return False
    return True
# test_math_operations.py
import pytest
from math_operations import add, subtract, multiply, divide, factorial, is_
prime
def test_add():
    """Тест сложения"""
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
def test_subtract():
    """Тест вычитания"""
    assert subtract(5, 3) == 2
    assert subtract(0, 5) == -5
def test_multiply():
    """Тест умножения"""
    assert multiply(4, 3) == 12
    assert multiply(-2, 3) == -6
def test_divide():
    """Тест деления"""
    assert divide(10, 2) == 5
    assert divide(7, 2) == 3.5
def test_divide_by_zero():
    """Тест деления на ноль"""
    with pytest.raises(ValueError, match="Division by zero"):
        divide(10, 0)
```

```
def test_factorial():
    """Тест факториала"""
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(5) == 120
# Обратите внимание: не все функции полностью протестированы
```

Запуск анализа покрытия

```
# Запуск тестов с анализом покрытия
coverage run -m pytest test_math_operations.py
# Просмотр отчета в консоли
coverage report
# Создание HTML-отчета
coverage html
```

Типы покрытия

Покрытие строк (Line Coverage)

```
# example_coverage.py
def process_data(data, validate=True):
    """Обработка данных с валидацией"""
    if validate:
        if not data:  # Строка 4
            raise ValueError("Data cannot be empty")  # Строка 5

        if not isinstance(data, list):  # Строка 7
            raise TypeError("Data must be a list")  # Строка 8

    result = []  # Строка 10
    for item in data:  # Строка 11
        if isinstance(item, int):  # Строка 12
            result.append(item * 2)  # Строка 13
        else:  # Строка 14
            result.append(str(item).upper())  # Строка 15

    return result  # Строка 17
# test_coverage.py
from example_coverage import process_data
def test_process_valid_data():
    """Тест обработки валидных данных"""
    data = [1, 2, "hello", 3]
    result = process_data(data)
    expected = [2, 4, "HELLO", 6]
    assert result == expected
def test_process_without_validation():
    """Тест обработки без валидации"""
    data = [1, "test"]
    result = process_data(data, validate=False)
    expected = [2, "TEST"]
    assert result == expected
```

```
# Этот тест покрывает не все ветки условий!
# Строки 5 и 8 не будут выполнены
```

Покрытие ветвей (Branch Coverage)

```
# Включение анализа ветвей
# coverage run --branch -m pytest test_coverage.py
def grade_calculator(score):
    """Вычисление оценки по баллам"""
    if score >= 90:
        grade = 'A'
    elif score >= 80:
        grade = 'B'
    elif score >= 70:
        grade = 'C'
    elif score >= 60:
        grade = 'D'
    else:
        grade = 'F'

    return grade
def test_all_grades():
    """Тест всех возможных оценок"""
    assert grade_calculator(95) == 'A'
    assert grade_calculator(85) == 'B'
    assert grade_calculator(75) == 'C'
    assert grade_calculator(65) == 'D'
    assert grade_calculator(45) == 'F'

    # Граничные случаи
    assert grade_calculator(90) == 'A'
    assert grade_calculator(89) == 'B'
    assert grade_calculator(60) == 'D'
    assert grade_calculator(59) == 'F'
```

Интерпретация результатов покрытия

Анализ отчета

```
# user_manager.py
class User:
    def __init__(self, username, email, age=None):
        self.username = username
        self.email = email
        self.age = age
        self.active = True

    def deactivate(self):
        """Деактивация пользователя"""
        self.active = False
```

```
    def update_email(self, new_email):
        """Обновление email-пользователя"""
        if '@' not in new_email:
            raise ValueError("Invalid email format")
        self.email = new_email

    def can_vote(self):
        """Проверка права голоса"""
        return self.age is not None and self.age >= 18
class UserManager:
    def __init__(self):
        self.users = {}

    def add_user(self, user):
        """Добавление пользователя"""
        if user.username in self.users:
            raise ValueError(f"User {user.username} already exists")
        self.users[user.username] = user

    def get_user(self, username):
        """Получение пользователя"""
        return self.users.get(username)

    def get_active_users(self):
        """Получение активных пользователей"""
        return [user for user in self.users.values() if user.active]

    def get_voting_eligible_users(self):
        """Получение пользователей с правом голоса"""
        return [user for user in self.users.values()
                if user.active and user.can_vote()]
# test_user_manager.py
import pytest
from user_manager import User, UserManager
class TestUser:
    def test_user_creation(self):
        """Тест создания пользователя"""
        user = User("alice", "alice@example.com", 25)
        assert user.username == "alice"
        assert user.email == "alice@example.com"
        assert user.age == 25
        assert user.active is True

    def test_user_deactivation(self):
        """Тест деактивации пользователя"""
        user = User("bob", "bob@example.com")
        user.deactivate()
        assert user.active is False

    def test_email_update_valid(self):
```

```
        """Тест обновления валидного email"""
        user = User("charlie", "charlie@example.com")
        user.update_email("new_charlie@example.com")
        assert user.email == "new_charlie@example.com"

    def test_email_update_invalid(self):
        """Тест обновления невалидного email"""
        user = User("dave", "dave@example.com")
        with pytest.raises(ValueError, match="Invalid email format"):
            user.update_email("invalid_email")
class TestUserManager:
    def test_add_and_get_user(self):
        """Тест добавления и получения пользователя"""
        manager = UserManager()
        user = User("alice", "alice@example.com")

        manager.add_user(user)
        retrieved_user = manager.get_user("alice")

        assert retrieved_user is user

    def test_get_active_users(self):
        """Тест получения активных пользователей"""
        manager = UserManager()
        user1 = User("alice", "alice@example.com")
        user2 = User("bob", "bob@example.com")

        manager.add_user(user1)
        manager.add_user(user2)

        user2.deactivate()

        active_users = manager.get_active_users()
        assert len(active_users) == 1
        assert active_users[0] is user1
# Некоторые методы и ветки не покрыты тестами:
# - User.can_vote() не тестируется полностью
# - UserManager.add_user() - случай дублирования пользователя
# - UserManager.get_voting_eligible_users() вообще не тестируется
```

Улучшение покрытия

```
# Дополнительные тесты для улучшения покрытия
class TestUserVoting:
    def test_can_vote_adult(self):
        """Тест права голоса для совершеннолетнего"""
        user = User("adult", "adult@example.com", 21)
        assert user.can_vote() is True

    def test_can_vote_minor(self):
        """Тест права голоса для несовершеннолетнего"""
```

```
        user = User("minor", "minor@example.com", 16)
        assert user.can_vote() is False

    def test_can_vote_no_age(self):
        """Тест права голоса при отсутствии возраста"""
        user = User("unknown_age", "unknown@example.com")
        assert user.can_vote() is False

    def test_can_vote_exactly_18(self):
        """Тест права голоса в 18 лет"""
        user = User("eighteen", "eighteen@example.com", 18)
        assert user.can_vote() is True
class TestUserManagerComplete:
    def test_add_duplicate_user(self):
        """Тест добавления дублирующего пользователя"""
        manager = UserManager()
        user1 = User("alice", "alice1@example.com")
        user2 = User("alice", "alice2@example.com")

        manager.add_user(user1)

        with pytest.raises(ValueError, match="User alice already exists"):
            manager.add_user(user2)

    def test_get_nonexistent_user(self):
        """Тест получения несуществующего пользователя"""
        manager = UserManager()
        user = manager.get_user("nonexistent")
        assert user is None

def test_get_voting_eligible_users(self):
        """Тест получения пользователей с правом голоса"""
        manager = UserManager()

        adult_active = User("adult", "adult@example.com", 25)
        minor_active = User("minor", "minor@example.com", 16)
        adult_inactive = User("inactive", "inactive@example.com", 30)
        adult_no_age = User("no_age", "noage@example.com")

        manager.add_user(adult_active)
        manager.add_user(minor_active)
        manager.add_user(adult_inactive)
        manager.add_user(adult_no_age)

        adult_inactive.deactivate()

        voting_users = manager.get_voting_eligible_users()

        # Только взрослый активный пользователь может голосовать
        assert len(voting_users) == 1
        assert voting_users[0] is adult_active
```

Настройка и конфигурация покрытия

Исключение файлов из анализа

```
# .coveragerc - файл конфигурации coverage
"""
[run]
source = .
omit =
    */tests/*
    */venv/*
    setup.py
    conftest.py
[report]
exclude_lines =
    pragma: no cover
    def __repr__
    raise AssertionError
    raise NotImplementedError
    if __name__ == .__main__.:
[html]
directory = htmlcov
"""
# example_exclusions.py
def debug_function():  # pragma: no cover
    """Функция только для отладки"""
    print("Debug information")
    return True
class ConfigManager:
    def __init__(self):
        self.config = {}

    def __repr__(self):  # Исключается автоматически
        return f"ConfigManager({self.config})"

    def load_config(self, filename):
        """Загрузка конфигурации"""
        try:
            with open(filename, 'r') as f:
                # Основная логика
                self.config = {"loaded": True}
        except FileNotFoundError:
           raise NotImplementedError("Config file not found")  # Исключается

        return self.config
if __name__ == "__main__":  # Исключается автоматически
    manager = ConfigManager()
    manager.load_config("config.json")
```

Интеграция с pytest

Использование pytest-cov

```
pip install pytest-cov
# Запуск тестов с покрытием через pytest
# pytest --cov=math_operations --cov-report=html
# pytest --cov=. --cov-report=term-missing
def test_integration_with_coverage():
    """Тест интеграции с системой покрытия"""
    from math_operations import add, is_prime

    # Тестируем различные функции
    assert add(1, 1) == 2
    assert is_prime(7) is True
    assert is_prime(4) is False
# conftest.py для автоматического запуска покрытия
import pytest
def pytest_configure(config):
    """Автоматическая настройка покрытия"""
    config.option.cov_source = ["."]
    config.option.cov_report = {"term-missing": True}
```

Метрики качества тестирования

Анализ качества покрытия

```
def analyze_coverage_quality():
    """Анализ качества покрытия кода"""

    coverage_metrics = {
        "line_coverage": 85,      # 85% строк покрыты
        "branch_coverage": 78,    # 78% ветвей покрыты
        "function_coverage": 92,  # 92% функций покрыты
    }

    # Рекомендации по качеству покрытия:
    recommendations = []

    if coverage_metrics["line_coverage"] < 80:
        recommendations.append("Increase line coverage to at least 80%")

    if coverage_metrics["branch_coverage"] < 75:
        recommendations.append("Improve branch coverage to at least 75%")

    if coverage_metrics["function_coverage"] < 90:
        recommendations.append("Ensure at least 90% of functions are tested")

    return recommendations
# Пример отчета покрытия
coverage_report = """
```

```
Name                  Stmts   Miss  Cover   Missing
-----------------------------------------------------
math_operations.py       45      8    82%   23-25, 45-48, 67
user_manager.py          32      3    91%   78-80
test_math.py             28      0   100%
-----------------------------------------------------
TOTAL                   105     11    90%
"""
def interpret_coverage_report(report):
    """Интерпретация отчета покрытия"""
    analysis = {
        "total_coverage": 90,
        "files_needing_attention": [
            {
                "file": "math_operations.py",
                "coverage": 82,
                "missing_lines": "23-25, 45-48, 67",
                "priority": "medium"
            }
        ],
        "well_tested_files": [
            {"file": "user_manager.py", "coverage": 91},
            {"file": "test_math.py", "coverage": 100}
        ]
    }

    return analysis
```

Лучшие практики покрытия

Покрытие кода — это полезная метрика, но важно помнить о ее ограничениях. **Высокое покрытие не гарантирует качество тестов**, а низкое покрытие не всегда означает плохие тесты.

Рекомендуемые цели покрытия:

- 80-90% для большинства проектов
- 95%+ для критически важного кода
- 70%+ для прототипов и экспериментов

Что не нужно тестировать:

- Тривиальные геттеры и сеттеры
- Код третьих библиотек
- Конфигурационные файлы

Фокус на качестве, а не количестве: лучше иметь меньше тестов, которые проверяют важную логику и граничные случаи, чем много тестов, покрывающих тривиальный код.

Покрытие кода тестами является важным инструментом контроля качества, но его следует использовать разумно, в сочетании с другими практиками обеспече-

ния качества. Главная цель — не достижение 100% покрытия, а создание надежных тестов, которые помогают предотвратить ошибки и облегчают сопровождение кода.

10.7. Документирование кода: docstring и типы

Качественная документация — это неотъемлемая часть профессионального программирования. Хорошо задокументированный код становится понятным не только автору через несколько месяцев, но и другим разработчикам. Python предоставляет встроенные механизмы для создания документации прямо в коде.

Строки документации (Docstrings)

Повторим, строки документации — это специальные строки, размещаемые в начале модулей, классов и функций. Они описывают назначение, параметры и возвращаемые значения. Стандарт PEP 257 определяет правила их оформления.

```
def calculate_area(width, height):
    """
    Вычисляет площадь прямоугольника.

    Args:
        width (float): Ширина прямоугольника
        height (float): Высота прямоугольника

    Returns:
        float: Площадь прямоугольника

    Raises:
        ValueError: Если width или height меньше или равны нулю
    """
    if width <= 0 or height <= 0:
        raise ValueError("Размеры должны быть положительными")
    return width * height
```

Для простых функций достаточно однострочного описания:

```
def greet(name):
    """Возвращает приветствие для указанного имени."""
    return f"Привет, {name}!"
```

Строки документации доступны через атрибут `__doc__` и функцию `help()`:

```
print(calculate_area.__doc__)
help(calculate_area)
```

Типизация в Python

Python является языком с динамической типизацией, но начиная с версии 3.5 поддерживает аннотации типов (type hints). Они не влияют на выполнение кода, но значительно улучшают его читаемость и позволяют использовать статические анализаторы.

```
from typing import List, Dict, Optional, Union
def process_data(numbers: List[int]) -> Dict[str, float]:
    """
    Обрабатывает список чисел и возвращает статистику.
    Args:
        numbers: Список целых чисел

    Returns:
        Словарь со статистическими показателями
    """
    return {
        "mean": sum(numbers) / len(numbers),
        "max": max(numbers),
        "min": min(numbers)
    }
def find_user(user_id: int) -> Optional[Dict[str, str]]:
    """
    Ищет пользователя по ID.

    Args:
        user_id: Идентификатор пользователя

    Returns:
        Словарь с данными пользователя или None, если не найден
    """
    # Имитация поиска в базе данных
    users_db = {1: {"name": "Alice", "email": "alice@example.com"}}
    return users_db.get(user_id)
```

Для более сложных случаев используются обобщенные типы:

```
from typing import TypeVar, Generic, Callable
T = TypeVar('T')
def apply_function(items: List[T], func: Callable[[T], T]) -> List[T]:
    """
    Применяет функцию к каждому элементу списка.

    Args:
        items: Список элементов любого типа
        func: Функция для применения к каждому элементу

    Returns:
        Список с результатами применения функции
    """
    return [func(item) for item in items]
```

Статический анализ с MyPy

MyPy — это статический анализатор типов для Python, который проверяет соответствие кода указанным аннотациям типов. Установка производится через pip:

```
pip install mypy
Пример проверки кода:
# math_utils.py
def divide(a: int, b: int) -> float:
    """Деление двух чисел."""
    return a / b
# Некорректное использование
result = divide("10", 5)  # MyPy выдаст ошибку
```

Запуск проверки:

```
mypy math_utils.py

MyPy также поддерживает конфигурацию через файл mypy.ini:

[mypy]
python_version = 3.9
warn_return_any = True
warn_unused_configs = True
disallow_untyped_defs = True
```

Документирование классов

Классы требуют особого подхода к документированию:

```
class BankAccount:
    """
    Класс для работы с банковским счетом.

    Attributes:
        account_number (str): Номер счета
        balance (float): Текущий баланс
        owner (str): Владелец счета
    """

     def __init__(self, account_number: str, owner: str, initial_balance:
float = 0.0):
        """
        Инициализация банковского счета.

        Args:
            account_number: Номер счета
            owner: Имя владельца
            initial_balance: Начальный баланс (по умолчанию 0.0)
        """
        self.account_number = account_number
        self.owner = owner
        self.balance = initial_balance

    def deposit(self, amount: float) -> None:
        """
        Пополнение счета.
```

```
        Args:
            amount: Сумма для пополнения

        Raises:
            ValueError: Если сумма меньше или равна нулю
        """
        if amount <= 0:
            raise ValueError("Сумма должна быть положительной")
        self.balance += amount
```

Сочетание хорошего документирования и типизации создает самодокументируемый код, который легко понимать, поддерживать и развивать.

10.8. Практическая работа: тестирование математической библиотеки

В этом разделе мы создадим математическую библиотеку и покроем ее комплексными тестами, применив методологию разработки через тестирование (Test-Driven Development, TDD). Этот подход предполагает написание тестов до создания самого кода.

Планирование математической библиотеки

Создадим библиотеку `mathlib.py` с базовыми математическими функциями:

```
# mathlib.py
"""
Математическая библиотека с базовыми функциями.
"""
import math
from typing import List, Union
class StatisticsError(Exception):
    """Исключение для статистических вычислений."""
    pass
def factorial(n: int) -> int:
    """
    Вычисляет факториал числа.

    Args:
        n: Неотрицательное целое число

    Returns:
        Факториал числа n

    Raises:
        ValueError: Если n отрицательное
        TypeError: Если n не целое число
    """
    if not isinstance(n, int):
        raise TypeError("Аргумент должен быть целым числом")
```

```
    if n < 0:
          raise ValueError("Факториал определен только для неотрицательных
чисел")
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

Разработка через тестирование (TDD)

Начнем с написания тестов для функции, которую еще не реализовали:

```
# test_mathlib.py
import unittest
import pytest
from mathlib import factorial, StatisticsError
class TestFactorial(unittest.TestCase):
    """Тесты для функции factorial."""

    def test_factorial_zero(self):
        """Факториал нуля равен единице."""
        self.assertEqual(factorial(0), 1)

    def test_factorial_positive_numbers(self):
        """Факториал положительных чисел."""
        self.assertEqual(factorial(1), 1)
        self.assertEqual(factorial(5), 120)
        self.assertEqual(factorial(10), 3628800)

    def test_factorial_negative_raises_error(self):
        """Факториал отрицательного числа вызывает ошибку."""
        with self.assertRaises(ValueError):
            factorial(-1)

        with self.assertRaises(ValueError):
            factorial(-10)

    def test_factorial_non_integer_raises_error(self):
        """Факториал не целого числа вызывает ошибку."""
        with self.assertRaises(TypeError):
            factorial(3.14)

        with self.assertRaises(TypeError):
            factorial("5")
```

Теперь добавим функцию для работы со статистикой:

```
# Добавляем в mathlib.py
def mean(numbers: List[Union[int, float]]) -> float:
    """
```

```
    Вычисляет среднее арифметическое.

    Args:
        numbers: Список чисел

    Returns:
        Среднее арифметическое

    Raises:
        StatisticsError: Если список пуст
        TypeError: Если элементы списка не числовые
    """
    if not numbers:
       raise StatisticsError("Нельзя вычислить среднее для пустого списка")

    for num in numbers:
        if not isinstance(num, (int, float)):
            raise TypeError("Все элементы должны быть числами")

    return sum(numbers) / len(numbers)
def median(numbers: List[Union[int, float]]) -> float:
    """
    Вычисляет медиану.

    Args:
        numbers: Список чисел

    Returns:
        Медиана

    Raises:
        StatisticsError: Если список пуст
    """
    if not numbers:
       raise StatisticsError("Нельзя вычислить медиану для пустого списка")

    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)

    if n % 2 == 1:
        return float(sorted_numbers[n // 2])
    else:
        mid1, mid2 = sorted_numbers[n // 2 - 1], sorted_numbers[n // 2]
        return (mid1 + mid2) / 2.0
```

Расширенное тестирование с Pytest

Создадим более подробные тесты с использованием `pytest`:

```
# test_statistics.py
```

```
import pytest
from mathlib import mean, median, StatisticsError
class TestMean:
    """Тесты для функции mean."""

    def test_mean_single_number(self):
        """Среднее одного числа равно самому числу."""
        assert mean([5]) == 5.0
        assert mean([0]) == 0.0
        assert mean([-3]) == -3.0

    def test_mean_multiple_numbers(self):
        """Среднее нескольких чисел."""
        assert mean([1, 2, 3]) == 2.0
        assert mean([1, 2, 3, 4]) == 2.5
        assert mean([-1, 0, 1]) == 0.0

    def test_mean_with_floats(self):
        """Среднее с вещественными числами."""
        assert abs(mean([1.5, 2.5, 3.5]) - 2.5) < 1e-10

    def test_mean_empty_list_raises_error(self):
        """Среднее пустого списка вызывает ошибку."""
        with pytest.raises(StatisticsError):
            mean([])

    def test_mean_non_numeric_raises_error(self):
        """Среднее со строками вызывает ошибку."""
        with pytest.raises(TypeError):
            mean([1, 2, "3"])
class TestMedian:
    """Тесты для функции median."""

    def test_median_odd_length(self):
        """Медиана списков нечетной длины."""
        assert median([1, 2, 3]) == 2.0
        assert median([5, 1, 3]) == 3.0

    def test_median_even_length(self):
        """Медиана списков четной длины."""
        assert median([1, 2]) == 1.5
        assert median([1, 2, 3, 4]) == 2.5

    @pytest.mark.parametrize("numbers,expected", [
        ([1], 1.0),
        ([1, 1, 1], 1.0),
        ([-1, 0, 1], 0.0),
        ([10, 20, 30, 40], 25.0),
    ])
    def test_median_parametrized(self, numbers, expected):
```

```
        """Параметризованные тесты медианы."""
        assert median(numbers) == expected
```

Интеграционные тесты и фикстуры

Создадим тесты, которые используют несколько функций вместе:

```
# test_integration.py
import pytest
from mathlib import mean, median, factorial
@pytest.fixture
def sample_data():
    """Фикстура с тестовыми данными."""
    return {
        'small_list': [1, 2, 3],
        'large_list': list(range(1, 101)),
        'mixed_list': [-5, 0, 3.14, 10, 15.5],
        'factorial_inputs': [0, 1, 5, 10]
    }
class TestIntegration:
    """Интеграционные тесты библиотеки."""

    def test_statistics_consistency(self, sample_data):
        """Проверка согласованности статистических функций."""
        numbers = sample_data['mixed_list']

        # Для симметричного распределения среднее близко к медиане
        symmetric_data = [-2, -1, 0, 1, 2]
        assert abs(mean(symmetric_data) - median(symmetric_data)) < 0.1

    def test_factorial_with_statistics(self, sample_data):
        """Тестирование факториала со статистическими данными."""
       factorials = [factorial(n) for n in sample_data['factorial_inputs']]

        # Факториалы должны быть возрастающей последовательностью
        assert factorials == sorted(factorials)

        # Среднее факториалов
        mean_factorial = mean(factorials)
        assert mean_factorial > 0
        assert isinstance(mean_factorial, float)
```

Тестирование производительности и граничных случаев

```
# test_performance.py
import time
import pytest
from mathlib import factorial, mean
```

```
class TestPerformance:
    """Тесты производительности."""

    def test_factorial_performance(self):
        """Факториал больших чисел должен вычисляться быстро."""
        start_time = time.time()
        result = factorial(100)
        end_time = time.time()

        assert end_time - start_time < 0.1  # Менее 100ms
        assert result > 0

    def test_mean_large_dataset(self):
        """Среднее для больших наборов данных."""
        large_dataset = list(range(100000))

        start_time = time.time()
        result = mean(large_dataset)
        end_time = time.time()

        assert end_time - start_time < 1.0  # Менее 1 секунды
        assert abs(result - 49999.5) < 0.1
class TestEdgeCases:
    """Тесты граничных случаев."""

    def test_very_large_numbers(self):
        """Работа с очень большими числами."""
        large_numbers = [10**10, 10**11, 10**12]
        result = mean(large_numbers)
        assert result > 0
        assert isinstance(result, float)

    def test_very_small_numbers(self):
        """Работа с очень малыми числами."""
        small_numbers = [1e-10, 1e-11, 1e-12]
        result = mean(small_numbers)
        assert result > 0
        assert result < 1e-9
```

Запуск всех тестов с отчетом о покрытии:

```
pytest --cov=mathlib --cov-report=html test_*.py
```

Этот комплексный подход к тестированию обеспечивает высокое качество кода и уверенность в его корректности. Методология TDD помогает создавать более продуманную архитектуру и выявлять потенциальные проблемы на раннем этапе разработки.

Глава 11. Работа с внешними данными

11.1. HTTP-запросы: взаимодействие с веб-сервисами

В современном программировании редко можно встретить приложение, которое работает в полной изоляции. Большинство программ взаимодействуют с внешними сервисами, получают данные из интернета или отправляют информацию на удаленные серверы. Протокол HTTP (HyperText Transfer Protocol) стал универсальным языком для такого взаимодействия.

Python предоставляет несколько способов работы с HTTP, но наиболее популярной и удобной библиотекой является **`requests`**. Эта библиотека превращает сложные операции сетевого взаимодействия в простые и понятные функции.

Установка и первые шаги

Библиотека `requests` не входит в стандартную поставку Python, поэтому ее необходимо установить:

```
pip install requests
```

Начнем с простейшего примера — получения веб-страницы:

```
import requests
# Выполняем GET-запрос к веб-странице
response = requests.get('https://httpbin.org/get')
# Проверяем статус ответа
print(f"Статус код: {response.status_code}")
# Получаем содержимое ответа как текст
print(f"Содержимое: {response.text[:200]}...")
```

В этом примере мы обращаемся к сервису `httpbin.org` — это специальный сервис для тестирования HTTP-запросов. Он возвращает информацию о запросе в формате JSON, что очень удобно для изучения.

Анатомия HTTP-запроса

HTTP-запрос состоит из нескольких компонентов: метода (GET, POST, PUT, DELETE), URL, заголовков и тела запроса. Библиотека `requests` позволяет легко управлять всеми этими элементами:

```
import requests
# GET-запрос с параметрами
params = {'key1': 'value1', 'key2': 'value2'}
response = requests.get('https://httpbin.org/get', params=params)
print(f"URL запроса: {response.url}")
print(f"Статус: {response.status_code}")
```

```
# Ответ в формате JSON
data = response.json()
print(f"Параметры запроса: {data['args']}")
```

Параметры автоматически кодируются и добавляются к URL. В результате URL будет выглядеть как `https://httpbin.org/get?key1=value1&key2=value2`.

Работа с различными HTTP-методами

HTTP предоставляет несколько методов для различных операций. GET используется для получения данных, POST — для отправки данных, PUT — для обновления, DELETE — для удаления:

```
import requests
import json
# POST-запрос с данными
post_data = {'username': 'user123', 'message': 'Привет, мир!'}
response = requests.post('https://httpbin.org/post', data=post_data)
print(f"Статус POST: {response.status_code}")
# POST-запрос с JSON
json_data = {'name': 'Python', 'version': '3.11'}
headers = {'Content-Type': 'application/json'}
response = requests.post('https://httpbin.org/post',
                         json=json_data,
                         headers=headers)
result = response.json()
print(f"Отправленные JSON данные: {result['json']}")
```

Обработка ответов сервера

Объект ответа (`response`) содержит всю необходимую информацию о результате запроса. Важно правильно интерпретировать полученные данные:

```
import requests
def analyze_response(url):
    """Анализирует HTTP-ответ и выводит подробную информацию"""

    response = requests.get(url)

    print(f"URL: {response.url}")
    print(f"Статус код: {response.status_code}")
    print(f"Заголовки ответа:")
    for key, value in response.headers.items():
        print(f"  {key}: {value}")

    # Проверяем тип содержимого
    content_type = response.headers.get('content-type', '').lower()

    if 'application/json' in content_type:
        try:
            json_data = response.json()
            print(f"JSON-данные: {json_data}")
```

```
        except ValueError:
            print("Ошибка при декодировании JSON")

    elif 'text/html' in content_type:
      print(f"HTML содержимое (первые 200 символов): {response.text[:200]}")

    else:
        print(f"Тип содержимого: {content_type}")
        print(f"Размер: {len(response.content)} байт")
# Тестируем функцию
analyze_response('https://httpbin.org/json')
```

Обработка ошибок и исключений

Сетевые операции часто завершаются ошибками: сервер недоступен, превышено время ожидания, неверный URL. Правильная обработка ошибок — ключевой навык при работе с внешними сервисами:

```
import requests
from requests.exceptions import ConnectionError, Timeout, RequestException
def safe_request(url, timeout=5):
    """Выполняет HTTP-запрос с обработкой ошибок"""

    try:
        response = requests.get(url, timeout=timeout)

        # Проверяем успешность запроса
         response.raise_for_status()  # Вызывает исключение для 4xx и 5xx
статусов

        return response

    except ConnectionError:
        print(f"Ошибка соединения с {url}")
        return None

    except Timeout:
        print(f"Превышено время ожидания для {url}")
        return None

    except requests.exceptions.HTTPError as e:
        print(f"HTTP-ошибка: {e}")
        return None

    except RequestException as e:
        print(f"Общая ошибка запроса: {e}")
        return None
# Тестируем с разными URL
urls = [
    'https://httpbin.org/get',
    'https://httpbin.org/status/404',
```

```
    'https://nonexistent-site-12345.com'
]
for url in urls:
    print(f"\nТестирование: {url}")
    result = safe_request(url)
    if result:
        print(f"Успех: статус {result.status_code}")
```

Работа с заголовками и аутентификацией

Многие веб-сервисы требуют специальных заголовков или аутентификации. Библиотека `requests` предоставляет удобные способы их настройки:

```
import requests
from requests.auth import HTTPBasicAuth
# Пользовательские заголовки
headers = {
    'User-Agent': 'My Python App 1.0',
    'Accept': 'application/json',
    'Authorization': 'Bearer my-secret-token'
}
response = requests.get('https://httpbin.org/headers', headers=headers)
result = response.json()
print("Отправленные заголовки:", result['headers'])
# HTTP Basic Authentication
response = requests.get('https://httpbin.org/basic-auth/user/pass',
                        auth=HTTPBasicAuth('user', 'pass'))
if response.status_code == 200:
    print("Аутентификация успешна")
    print(response.json())
else:
    print("Ошибка аутентификации")
```

Сессии и постоянные соединения

Когда нужно выполнить несколько запросов к одному серверу, эффективнее использовать сессии. Они позволяют переиспользовать соединения и автоматически сохранять `cookies`:

```
import requests
# Создаем сессию
session = requests.Session()
# Устанавливаем общие заголовки для всех запросов в сессии
session.headers.update({'User-Agent': 'My App 1.0'})
# Выполняем несколько запросов в рамках одной сессии
response1 = session.get('https://httpbin.org/cookies/set/session_id/12345')
response2 = session.get('https://httpbin.org/cookies')
print("Cookies в сессии:", response2.json())
# Не забываем закрыть сессию
session.close()
```

Практический пример: мониторинг веб-сайта

Создадим полезное приложение, которое проверяет доступность веб-сайтов и анализирует время ответа:

```
import requests
import time
from datetime import datetime
class WebsiteMonitor:
    """Класс для мониторинга доступности веб-сайтов"""

    def __init__(self, timeout=10):
        self.timeout = timeout
        self.session = requests.Session()
        # Устанавливаем реалистичный User-Agent
        self.session.headers.update({
            'User-Agent': 'Mozilla/5.0 (Website Monitor Bot)'
        })

    def check_website(self, url):
        """Проверяет один веб-сайт и возвращает статистику"""

        start_time = time.time()

        try:
            response = self.session.get(url, timeout=self.timeout)
            end_time = time.time()

            return {
                'url': url,
                'status': 'UP',
                'status_code': response.status_code,
                'response_time': round((end_time - start_time) * 1000, 2),
                'content_length': len(response.content),
                'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            }

        except Exception as e:
            end_time = time.time()

            return {
                'url': url,
                'status': 'DOWN',
                'error': str(e),
                'response_time': round((end_time - start_time) * 1000, 2),
                'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            }

    def monitor_websites(self, urls, interval=60):
        """Мониторит список сайтов с заданным интервалом"""

        print(f"Начинаем мониторинг {len(urls)} сайтов...")
```

```
        print(f"Интервал проверки: {interval} секунд\n")

        try:
            while True:
               print(f"=== Проверка в {datetime.now().strftime('%H:%M:%S')}
===")

                for url in urls:
                    result = self.check_website(url)

                    if result['status'] == 'UP':
                        print(f"✓ {url} - {result['status_code']} "
                              f"({result['response_time']}ms)")
                    else:
                        print(f"✗ {url} - {result['error']}")

                print(f"\nСледующая проверка через {interval} секунд...\n")
                time.sleep(interval)

        except KeyboardInterrupt:
            print("\nМониторинг остановлен пользователем")

        finally:
            self.session.close()
# Использование монитора
if __name__ == "__main__":
    monitor = WebsiteMonitor()

    sites_to_check = [
        'https://httpbin.org',
        'https://python.org',
        'https://github.com'
    ]

    # Однократная проверка
    print("Однократная проверка сайтов:")
    for site in sites_to_check:
        result = monitor.check_website(site)
        print(f"{site}: {result}")
```

11.2. Парсинг HTML: библиотека BeautifulSoup

Получение HTML-кода страницы — это только первый шаг. Чтобы извлечь из нее полезную информацию, нужно уметь анализировать структуру HTML-документа. Библиотека **`BeautifulSoup`** превращает хаотичный HTML в удобную для работы структуру данных.

HTML-документ имеет древовидную структуру, называемую DOM (Document Object Model). BeautifulSoup позволяет навигировать по этому дереву, искать нужные элементы и извлекать данные с помощью интуитивно понятного API.

Установка и первое знакомство

BeautifulSoup требует дополнительного парсера HTML. Рекомендуется использовать lxml для лучшей производительности:

```
pip install beautifulsoup4 lxml
Начнем с простого примера парсинга HTML:
from bs4 import BeautifulSoup
import requests
# Получаем HTML-страницу
response = requests.get('https://httpbin.org/html')
html_content = response.text
# Создаем объект BeautifulSoup
soup = BeautifulSoup(html_content, 'lxml')
# Выводим HTML
print("Структура документа:")
print(soup.prettify())
# Извлекаем заголовок страницы
title = soup.find('title')
if title:
    print(f"\nЗаголовок: {title.text}")
# Находим все ссылки
links = soup.find_all('a')
print(f"\nНайдено ссылок: {len(links)}")
for link in links:
    href = link.get('href')
    text = link.text.strip()
    print(f"  {text} -> {href}")
```

Навигация по DOM-дереву

BeautifulSoup предоставляет несколько способов поиска элементов в HTML-документе. Каждый метод имеет свои преимущества в определенных ситуациях:

```
from bs4 import BeautifulSoup
# Пример HTML для демонстрации
html_example = """
<!DOCTYPE html>
<html>
<head>
    <title>Пример страницы</title>
</head>
<body>
    <div class="header">
        <h1 id="main-title">Добро пожаловать</h1>
        <nav>
            <ul>
                <li><a href="/home">Главная</a></li>
                <li><a href="/about">О нас</a></li>
                <li><a href="/contact">Контакты</a></li>
            </ul>
        </nav>
```

```
    </div>
    <main class="content">
        <article class="post">
            <h2>Первая статья</h2>
            <p>Содержимое первой статьи...</p>
            <div class="meta">
                <span class="author">Автор: Иван</span>
                <span class="date">2023-11-15</span>
            </div>
        </article>
        <article class="post">
            <h2>Вторая статья</h2>
            <p>Содержимое второй статьи...</p>
            <div class="meta">
                <span class="author">Автор: Мария</span>
                <span class="date">2023-11-16</span>
            </div>
        </article>
    </main>
</body>
</html>
"""
soup = BeautifulSoup(html_example, 'lxml')
# 1. Поиск по тегу
print("=== Поиск по тегу ===")
first_h2 = soup.find('h2')  # Находит первый элемент h2
all_h2 = soup.find_all('h2')  # Находит все элементы h2
print(f"Первый h2: {first_h2.text}")
print(f"Все h2: {[h.text for h in all_h2]}")
# 2. Поиск по атрибутам
print("\n=== Поиск по атрибутам ===")
main_title = soup.find('h1', id='main-title')
articles = soup.find_all('article', class_='post')
print(f"Главный заголовок: {main_title.text}")
print(f"Количество статей: {len(articles)}")
# 3. Поиск по CSS-селекторам
print("\n=== CSS-селекторы ===")
authors = soup.select('.meta .author')
dates = soup.select('span.date')
nav_links = soup.select('nav a')
print("Авторы:", [author.text for author in authors])
print("Даты:", [date.text for date in dates])
print("Навигация:", [link.text for link in nav_links])
```

CSS-селекторы: мощный инструмент поиска

CSS-селекторы предоставляют гибкий и интуитивно понятный способ поиска элементов. Они особенно полезны, когда нужно найти элементы по сложным критериям:

```
from bs4 import BeautifulSoup
```

```
# Более сложный пример HTML
complex_html = """
<div class="container">
    <div class="sidebar">
        <h3>Категории</h3>
        <ul>
            <li><a href="/tech" class="category-link">Технологии</a></li>
            <li><a href="/science" class="category-link">Наука</a></li>
        </ul>
    </div>
    <div class="content">
        <article data-category="tech" class="featured">
            <h2>Новости технологий</h2>
            <p>Статья о последних достижениях...</p>
            <footer>
                <span class="views">1000 просмотров</span>
                <span class="likes">50 лайков</span>
            </footer>
        </article>
        <article data-category="science">
            <h2>Научные открытия</h2>
            <p>Интересные факты из мира науки...</p>
            <footer>
                <span class="views">750 просмотров</span>
                <span class="likes">30 лайков</span>
            </footer>
        </article>
    </div>
</div>
"""
soup = BeautifulSoup(complex_html, 'lxml')
# Различные CSS-селекторы
selectors_examples = {
    # Прямой потомок
    '.container > .content': 'Прямые потомки .container с классом .content',

    # Потомок любого уровня
    '.container .category-link': 'Все ссылки категорий внутри контейнера',

    # Атрибуты
    'article[data-category="tech"]': 'Статьи с категорией "tech"',

    # Псевдоклассы
    'article:first-child': 'Первая статья',

    # Соседние элементы
    'h2 + p': 'Параграфы, следующие сразу за заголовком h2',

    # Множественный выбор
    '.views, .likes': 'Элементы с классами views или likes'
}
```

```
for selector, description in selectors_examples.items():
    elements = soup.select(selector)
    print(f"{description}:")
    print(f"  Селектор: {selector}")
    print(f"  Найдено: {len(elements)} элементов")
    for elem in elements:
        print(f"    {elem.name}: {elem.get_text(strip=True)}")
    print()
```

Извлечение данных и атрибутов

BeautifulSoup предоставляет множество способов извлечения информации из найденных элементов:

```
from bs4 import BeautifulSoup
import requests
def extract_article_data(article_element):
    """Извлекает структурированные данные из элемента статьи"""

    data = {}

    # Заголовок
    title_elem = article_element.find('h2')
     data['title'] = title_elem.text.strip() if title_elem else 'Без заго-
ловка'

    # Содержимое
    content_elem = article_element.find('p')
    data['content'] = content_elem.text.strip() if content_elem else ''

    # Метаданные
    meta_elem = article_element.find('footer')
    if meta_elem:
        views_elem = meta_elem.find(class_='views')
        likes_elem = meta_elem.find(class_='likes')

        # Извлекаем числа из текста
       data['views'] = int(views_elem.text.split()[0]) if views_elem else 0
       data['likes'] = int(likes_elem.text.split()[0]) if likes_elem else 0

    # Атрибуты
    data['category'] = article_element.get('data-category', 'unknown')
    data['featured'] = 'featured' in article_element.get('class', [])

    return data
# Используем предыдущий HTML
soup = BeautifulSoup(complex_html, 'lxml')
articles = soup.find_all('article')
print("Извлеченные данные статей:")
```

```
for i, article in enumerate(articles, 1):
    data = extract_article_data(article)
    print(f"\nСтатья {i}:")
    for key, value in data.items():
        print(f"  {key}: {value}")
```

Обработка текста и очистка данных

Реальные веб-страницы часто содержат лишние пробелы, специальные символы и другой «мусор». `BeautifulSoup` предоставляет инструменты для очистки данных:

```
from bs4 import BeautifulSoup
import re
# HTML с различными проблемами форматирования
messy_html = """
<div class="article">
    <h1>   Заголовок с лишними пробелами   </h1>
    <p>
        Параграф с
        переносами строк и
        <span>   лишними   </span> пробелами.
    </p>
    <div class="content">
        Полезный контент
        <script>alert('Вредный скрипт')</script>
        <style>.hidden { display: none; }</style>
        Еще полезный контент
    </div>
    <footer>
        &copy; 2023 | Все права защищены
    </footer>
</div>
"""
soup = BeautifulSoup(messy_html, 'lxml')
def clean_text(element):
    """Очищает текст от лишних пробелов и специальных символов"""

    # Удаляем скрипты и стили
    for script in element(["script", "style"]):
        script.decompose()

    # Получаем текст
    text = element.get_text()

    # Убираем лишние пробелы и переносы строк
    text = re.sub(r'\s+', ' ', text).strip()

    return text
def extract_clean_data(soup):
    """Извлекает и очищает данные из HTML"""
```

```
    article = soup.find('div', class_='article')
    if not article:
        return None
    # Извлекаем заголовок
    title_elem = article.find('h1')
    title = title_elem.get_text(strip=True) if title_elem else ''

    # Извлекаем и очищаем содержимое
    content_elem = article.find('div', class_='content')
    content = clean_text(content_elem) if content_elem else ''

    # Извлекаем параграф
    p_elem = article.find('p')
    paragraph = clean_text(p_elem) if p_elem else ''

    # Извлекаем footer
    footer_elem = article.find('footer')
    footer = clean_text(footer_elem) if footer_elem else ''

    return {
        'title': title,
        'paragraph': paragraph,
        'content': content,
        'footer': footer
    }
# Демонстрация очистки
print("Исходный HTML:")
print(soup.prettify())
print("\nИзвлеченные и очищенные данные:")
clean_data = extract_clean_data(soup)
for key, value in clean_data.items():
    print(f"{key}: '{value}'")
```

Практический пример: парсинг новостного сайта

Создадим реальное приложение для извлечения новостей с веб-сайта:

```
from bs4 import BeautifulSoup
import requests
from datetime import datetime
import time
class NewsParser:
    """Парсер новостей с веб-сайтов"""

    def __init__(self, base_url, headers=None):
        self.base_url = base_url
        self.session = requests.Session()

        # Устанавливаем заголовки для имитации браузера
        default_headers = {
```

```
            'User-Agent': 'Mozilla/5.0 (compatible; NewsParser/1.0)'
        }
        if headers:
            default_headers.update(headers)
        self.session.headers.update(default_headers)

    def parse_article_list(self, url, article_selector, title_selector,
                           link_selector=None):
        """Парсит список статей со страницы"""

        try:
            response = self.session.get(url, timeout=10)
            response.raise_for_status()

            soup = BeautifulSoup(response.text, 'lxml')
            articles = soup.select(article_selector)

            parsed_articles = []

            for article in articles:
                # Извлекаем заголовок
                title_elem = article.select_one(title_selector)
                if not title_elem:
                    continue

                title = title_elem.get_text(strip=True)

                # Извлекаем ссылку
                if link_selector:
                    link_elem = article.select_one(link_selector)
                    link = link_elem.get('href') if link_elem else None
                else:
                    # Пытаемся найти ссылку в заголовке
                    link = title_elem.get('href') if title_elem.name == 'a'
else None
                    if not link:
                        # Ищем ссылку внутри заголовка
                        link_elem = title_elem.find('a')
                        link = link_elem.get('href') if link_elem else None

                # Делаем ссылку абсолютной
                if link and link.startswith('/'):
                    link = self.base_url + link

                parsed_articles.append({
                    'title': title,
                    'link': link,
                    'parsed_at': datetime.now().isoformat()
                })
```

```
            return parsed_articles

        except requests.exceptions.RequestException as e:
            print(f"Ошибка при загрузке страницы: {e}")
            return []
        except Exception as e:
            print(f"Ошибка при парсинге: {e}")
            return []

    def parse_article_content(self, url, content_selector):
        """Парсит содержимое отдельной статьи"""

        try:
            response = self.session.get(url, timeout=10)
            response.raise_for_status()

            soup = BeautifulSoup(response.text, 'lxml')

            # Извлекаем содержимое
            content_elem = soup.select_one(content_selector)
            if not content_elem:
                return None

            # Удаляем скрипты и стили
            for unwanted in content_elem(['script', 'style', 'iframe']):
                unwanted.decompose()

            # Извлекаем текст и очищаем его
            content = content_elem.get_text(separator=' ', strip=True)

            return {
                'content': content,
                'parsed_at': datetime.now().isoformat()
            }

        except Exception as e:
            print(f"Ошибка при парсинге статьи {url}: {e}")
            return None
# Пример использования для гипотетического новостного сайта
def demo_news_parsing():
    """Демонстрация парсинга новостей"""

    # Примечание: это демонстрационный пример
    # В реальности нужно изучить структуру конкретного сайта

    parser = NewsParser('https://example-news-site.com')

    # Парсим список статей
    print("Парсинг списка новостей...")
    articles = parser.parse_article_list(
```

```
        url='https://example-news-site.com/news',
        article_selector='.news-item',
        title_selector='h3 a',
        link_selector='h3 a'
    )
    print(f"Найдено {len(articles)} новостей:")

    # Выводим первые несколько новостей
    for i, article in enumerate(articles[:5], 1):
        print(f"{i}. {article['title']}")
        print(f"   Ссылка: {article['link']}")

        # Парсим содержимое первой статьи
        if i == 1 and article['link']:
            print("   Парсинг содержимого...")
            content_data = parser.parse_article_content(
                article['link'],
                '.article-content'
            )

            if content_data:
                # Показываем первые 200 символов
                preview = content_data['content'][:200] + "..."
                print(f"   Содержимое: {preview}")

        print()
# Функция для безопасного тестирования
def safe_parsing_demo():
    """Демонстрация парсинга с реальным сайтом для тестирования"""

    # Используем httpbin для демонстрации
    html_content = """
    <html>
    <body>
        <div class="news-container">
            <article class="news-item">
                <h3><a href="/news/1">Первая новость</a></h3>
                <p>Краткое описание первой новости...</p>
            </article>
            <article class="news-item">
                <h3><a href="/news/2">Вторая новость</a></h3>
                <p>Краткое описание второй новости...</p>
            </article>
        </div>
    </body>
    </html>
    """

    soup = BeautifulSoup(html_content, 'lxml')
```

```
    # Парсим статьи
    articles = soup.select('.news-item')

    print("Демонстрация парсинга HTML:")
    for i, article in enumerate(articles, 1):
        title_elem = article.select_one('h3 a')
        desc_elem = article.select_one('p')

        title = title_elem.get_text(strip=True) if title_elem else 'Без за-
головка'
        link = title_elem.get('href') if title_elem else None
        description = desc_elem.get_text(strip=True) if desc_elem else ''

        print(f"{i}. {title}")
        print(f"   Ссылка: {link}")
        print(f"   Описание: {description}")
        print()
if __name__ == "__main__":
    safe_parsing_demo()
```

Работа с формами и интерактивными элементами

BeautifulSoup также может анализировать HTML-формы, что полезно для автоматизации взаимодействия с веб-сайтами:

```
from bs4 import BeautifulSoup
# HTML с формой
form_html = """
<form id="contact-form" action="/submit" method="post">
    <div class="form-group">
        <label for="name">Имя:</label>
        <input type="text" id="name" name="name" required>
    </div>
    <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
    </div>
    <div class="form-group">
        <label for="category">Категория:</label>
        <select name="category" id="category">
            <option value="general">Общие вопросы</option>
            <option value="technical" selected>Технические вопросы</option>
            <option value="billing">Вопросы по оплате</option>
        </select>
    </div>
    <div class="form-group">
        <label for="message">Сообщение:</label>
        <textarea name="message" id="message" rows="5" placeholder="Введите
ваше сообщение..."></textarea>
    </div>
```

```
    <div class="form-group">
        <input type="checkbox" name="newsletter" id="newsletter" checked>
        <label for="newsletter">Подписаться на рассылку</label>
    </div>
    <button type="submit">Отправить</button>
</form>
"""
def analyze_form(html_content):
    """Анализирует HTML-форму и извлекает информацию о полях"""

    soup = BeautifulSoup(html_content, 'lxml')
    forms = soup.find_all('form')

    form_data = []

    for form in forms:
        form_info = {
            'id': form.get('id'),
            'action': form.get('action'),
            'method': form.get('method', 'get').upper(),
            'fields': []
        }

        # Анализируем поля ввода
        inputs = form.find_all(['input', 'select', 'textarea'])

        for field in inputs:
            field_info = {
                'type': field.get('type', field.name),
                'name': field.get('name'),
                'id': field.get('id'),
                'required': field.has_attr('required'),
                'placeholder': field.get('placeholder', ''),
                'value': field.get('value', '')
            }

            # Специальная обработка для разных типов полей
            if field.name == 'select':
                options = field.find_all('option')
                field_info['options'] = []

                for option in options:
                    opt_info = {
                        'value': option.get('value', ''),
                        'text': option.get_text(strip=True),
                        'selected': option.has_attr('selected')
                    }
                    field_info['options'].append(opt_info)
```

```
            elif field.get('type') == 'checkbox' or field.get('type') ==
'radio':
                field_info['checked'] = field.has_attr('checked')

            form_info['fields'].append(field_info)

        form_data.append(form_info)

    return form_data
# Анализируем форму
print("Анализ HTML-формы:")
forms = analyze_form(form_html)
for form in forms:
    print(f"\nФорма: {form['id']}")
    print(f"Действие: {form['action']}")
    print(f"Метод: {form['method']}")
    print(f"Полей: {len(form['fields'])}")

    for field in form['fields']:
        print(f"\n  Поле: {field['name']} ({field['type']})")
        if field['required']:
            print("    Обязательное поле")
        if field['placeholder']:
            print(f"    Подсказка: {field['placeholder']}")
        if field.get('options'):
            print("    Варианты:")
            for option in field['options']:
                marker = " [выбран]" if option['selected'] else ""
                print(f"      {option['value']}: {option['text']}{marker}")
        if field.get('checked'):
            print("    Отмечен по умолчанию")
```

Лучшие практики парсинга

При работе с реальными веб-сайтами важно следовать определенным правилам, чтобы избежать проблем и создать надежные решения:

```
from bs4 import BeautifulSoup
import requests
import time
import random
from urllib.robotparser import RobotFileParser
class RespectfulParser:
    """Парсер, который уважает правила сайта"""

    def __init__(self, base_url, user_agent=None, delay_range=(1, 3)):
        self.base_url = base_url
        self.delay_range = delay_range
        self.session = requests.Session()
```

```
        # Устанавливаем разумный User-Agent
        ua = user_agent or 'Mozilla/5.0 (compatible; RespectfulParser/1.0)'
        self.session.headers.update({'User-Agent': ua})

        # Проверяем robots.txt
        self.check_robots_txt()

    def check_robots_txt(self):
        """Проверяет robots.txt сайта"""
        try:
            robots_url = f"{self.base_url}/robots.txt"
            rp = RobotFileParser()
            rp.set_url(robots_url)
            rp.read()

            user_agent = self.session.headers.get('User-Agent', '*')

            print(f"Robots.txt для {self.base_url}:")
               print(f"Можно парсить: {rp.can_fetch(user_agent, self.base_
url)}")

            # Получаем задержку, если указана
            crawl_delay = rp.crawl_delay(user_agent)
            if crawl_delay:
                print(f"Рекомендуемая задержка: {crawl_delay} секунд")
                self.delay_range = (crawl_delay, crawl_delay + 1)

        except Exception as e:
            print(f"Не удалось проверить robots.txt: {e}")

    def respectful_get(self, url, **kwargs):
        """Выполняет GET-запрос с задержкой"""

        # Случайная задержка между запросами
        delay = random.uniform(*self.delay_range)
        time.sleep(delay)

        try:
            response = self.session.get(url, timeout=10, **kwargs)
            response.raise_for_status()
            return response

        except requests.exceptions.RequestException as e:
            print(f"Ошибка при запросе {url}: {e}")
            return None

    def parse_with_fallback(self, url, selectors_dict):
        """Парсит страницу с несколькими вариантами селекторов"""
```

```
        response = self.respectful_get(url)
        if not response:
            return None

        soup = BeautifulSoup(response.text, 'lxml')
        results = {}

        for key, selectors in selectors_dict.items():
            # selectors может быть строкой или списком строк
            if isinstance(selectors, str):
                selectors = [selectors]
            element = None
            for selector in selectors:
                element = soup.select_one(selector)
                if element:
                    break

            if element:
                results[key] = element.get_text(strip=True)
            else:
                results[key] = None
                print(f"Не найден элемент для {key} с селекторами: {selectors}")

        return results
# Пример использования
def demo_respectful_parsing():
    """Демонстрация уважительного парсинга"""

    parser = RespectfulParser('https://httpbin.org')

   # Определяем несколько вариантов селекторов на случай изменения структуры
    selectors = {
        'title': ['title', 'h1', '.main-title'],
        'description': ['.description', '.lead', 'p:first-of-type'],
        'content': ['.content', '.main-content', 'main']
    }

         result  =  parser.parse_with_fallback('https://httpbin.org/html',
selectors)

    if result:
        print("Результаты парсинга:")
        for key, value in result.items():
            status = "✓" if value else "✗"
            print(f"{status} {key}: {value}")
    else:
        print("Парсинг не удался")
if __name__ == "__main__":
    demo_respectful_parsing()
```

BeautifulSoup в сочетании с requests предоставляет мощные возможности для извлечения данных из веб-страниц. Основные принципы успешного парсинга включают понимание HTML-структуры, использование подходящих селекторов, обработку ошибок и соблюдение этики веб-скрапинга.

Важно помнить, что структура веб-сайтов может изменяться, поэтому надежные парсеры должны быть гибкими и включать механизмы обработки изменений. Всегда проверяйте robots.txt сайта и соблюдайте разумные интервалы между запросами, чтобы не создавать чрезмерную нагрузку на серверы.

11.3. Работа с API: получение данных из интернета

Современные веб-приложения редко работают изолированно — они постоянно обмениваются данными с другими сервисами через интерфейсы программирования приложений (API, Application Programming Interface). В отличие от веб-скрапинга, где мы извлекаем данные из HTML-страниц, API предоставляет структурированный способ доступа к информации.

REST API и основные принципы

Большинство современных веб-API построены по принципам REST (Representational State Transfer) — архитектурного стиля, который использует стандартные HTTP-методы для взаимодействия с ресурсами. Каждый ресурс имеет уникальный URL, а операции с ним выполняются через методы GET, POST, PUT и DELETE.

Рассмотрим практический пример работы с публичным API сервиса погоды:

```
import requests
import json
def get_weather_data(city, api_key):
    """Получение данных о погоде через API OpenWeatherMap"""
    base_url = "http://api.openweathermap.org/data/2.5/weather"

    # Параметры запроса
    params = {
        'q': city,
        'appid': api_key,
        'units': 'metric',  # температура в Цельсиях
        'lang': 'ru'        # описание на русском языке
    }

    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()  # проверка на HTTP-ошибки

        # API возвращает данные в формате JSON
        weather_data = response.json()
        return weather_data
```

```
    except requests.exceptions.RequestException as e:
        print(f"Ошибка при запросе к API: {e}")
        return None
```

Этот пример демонстрирует типичную структуру взаимодействия с API: формирование URL с параметрами, отправку запроса и обработку JSON-ответа.

Работа с JSON-данными

JSON (JavaScript Object Notation) стал стандартом для обмена данными между веб-сервисами. Python имеет встроенную поддержку этого формата через модуль `json`, который автоматически преобразует JSON-строки в словари и списки Python.

```
def parse_weather_response(weather_data):
    """Извлечение нужной информации из ответа API"""
    if not weather_data:
        return None

    # Извлекаем основную информацию
    city = weather_data['name']
    country = weather_data['sys']['country']
    temperature = weather_data['main']['temp']
    feels_like = weather_data['main']['feels_like']
    humidity = weather_data['main']['humidity']
    description = weather_data['weather'][0]['description']

    # Формируем удобную структуру данных
    weather_info = {
        'location': f"{city}, {country}",
        'temperature': temperature,
        'feels_like': feels_like,
        'humidity': humidity,
        'description': description.capitalize()
    }

    return weather_info
def display_weather(weather_info):
    """Отображение информации о погоде"""
    if not weather_info:
        print("Данные о погоде недоступны")
        return

    print(f"Погода в {weather_info['location']}:")
    print(f"Температура: {weather_info['temperature']}°C")
    print(f"Ощущается как: {weather_info['feels_like']}°C")
    print(f"Влажность: {weather_info['humidity']}%")
    print(f"Описание: {weather_info['description']}")
```

Важно всегда проверять структуру полученных JSON-данных, поскольку API может возвращать различные форматы в зависимости от успешности запроса или типа ошибки.

Аутентификация и API-ключи

Большинство API требует аутентификации (authentication) для контроля доступа и предотвращения злоупотреблений. Наиболее распространенные способы — API-ключи, токены и OAuth.

```
import os
from dotenv import load_dotenv
# Загружаем переменные окружения из файла .env
load_dotenv()
class WeatherAPI:
    """Класс для работы с API погоды"""
        def __init__(self):
        # Получаем API-ключ из переменных окружения
        self.api_key = os.getenv('OPENWEATHER_API_KEY')
        if not self.api_key:
             raise ValueError("API-ключ не найден. Установите OPENWEATHER_
API_KEY")

        self.base_url = "http://api.openweathermap.org/data/2.5/weather"

    def get_current_weather(self, city):
        """Получение текущей погоды для указанного города"""
        params = {
            'q': city,
            'appid': self.api_key,
            'units': 'metric',
            'lang': 'ru'
        }

        try:
          response = requests.get(self.base_url, params=params, timeout=10)
            response.raise_for_status()
            return response.json()

        except requests.exceptions.Timeout:
            print("Превышен лимит времени ожидания")
            return None
        except requests.exceptions.HTTPError as e:
            if response.status_code == 401:
                print("Ошибка аутентификации: проверьте API-ключ")
            elif response.status_code == 404:
                print(f"Город '{city}' не найден")
            else:
                print(f"HTTP ошибка: {e}")
            return None
```

Хранение API-ключей в переменных окружения — важная практика безопасности, которая предотвращает случайную публикацию секретных данных в репозиториях кода.

Пагинация: работа с большими наборами данных

Многие API не возвращают все данные сразу, а используют пагинацию (pagination) — разбиение результатов на страницы. Это позволяет эффективно обрабатывать большие объемы информации.

```
import time
class GitHubAPI:
    """Класс для работы с API GitHub"""

    def __init__(self, token=None):
        self.base_url = "https://api.github.com"
        self.session = requests.Session()

        if token:
            self.session.headers.update({
                'Authorization': f'token {token}',
                'Accept': 'application/vnd.github.v3+json'
            })

    def get_user_repos(self, username, per_page=30):
        """Получение всех репозиториев пользователя с пагинацией"""
        repos = []
        page = 1

        while True:
            url = f"{self.base_url}/users/{username}/repos"
            params = {
                'page': page,
                'per_page': per_page,
                'sort': 'updated',
                'direction': 'desc'
            }

            response = self.session.get(url, params=params)

            if response.status_code == 404:
                print(f"Пользователь '{username}' не найден")
                break

            response.raise_for_status()
            page_data = response.json()

            # Если страница пустая, значит данные закончились
            if not page_data:
```

```
                break
            repos.extend(page_data)
            print(f"Загружена страница {page}, получено {len(page_data)} ре-
позиториев")

            page += 1

            # Небольшая пауза между запросами из вежливости к API
            time.sleep(0.1)

        return repos
```

Этот пример показывает типичный паттерн работы с пагинированными API: цикл по страницам до получения пустого результата.

Rate Limiting: ограничения скорости запросов

API-сервисы устанавливают ограничения на количество запросов для предотвращения перегрузки серверов. Rate limiting может быть основан на количестве запросов в минуту, час или день:

```
import time
from datetime import datetime, timedelta
class RateLimitedAPI:
    """Базовый класс для работы с API с ограничениями скорости"""

    def __init__(self, requests_per_minute=60):
        self.requests_per_minute = requests_per_minute
        self.min_interval = 60.0 / requests_per_minute
        self.last_request_time = None
        self.request_count = 0
        self.session = requests.Session()

    def make_request(self, url, **kwargs):
        """Выполнение запроса с соблюдением ограничений скорости"""
        self._wait_if_needed()

        try:
            response = self.session.get(url, **kwargs)
            self._update_request_stats(response)
            return response

        except requests.exceptions.RequestException as e:
            print(f"Ошибка запроса: {e}")
            return None

    def _wait_if_needed(self):
        """Пауза при необходимости соблюдения ограничений"""
        if self.last_request_time:
            elapsed = time.time() - self.last_request_time
```

```
            if elapsed < self.min_interval:
                sleep_time = self.min_interval - elapsed
                print(f"Ожидание {sleep_time:.1f} сек. для соблюдения rate
limit")
                time.sleep(sleep_time)

    def _update_request_stats(self, response):
        """Обновление статистики запросов"""
        self.last_request_time = time.time()
        self.request_count += 1

        # Проверяем заголовки ответа на информацию о лимитах
        remaining = response.headers.get('X-RateLimit-Remaining')
        reset_time = response.headers.get('X-RateLimit-Reset')

        if remaining:
            print(f"Осталось запросов: {remaining}")
         if response.status_code == 429:  # Too Many Requests
            print("Превышен лимит запросов. Увеличиваем паузы...")
            self.min_interval *= 2  # Удваиваем интервал
```

Обработка ошибок API

Различные API могут возвращать ошибки в разных форматах. Важно предусмотреть обработку типичных ситуаций:

```
def handle_api_response(response):
    """Универсальная обработка ответов API"""
    try:
        # Проверяем статус-код
        if response.status_code == 200:
            return response.json()

        elif response.status_code == 401:
           raise APIError("Ошибка аутентификации: проверьте токен или ключ")

        elif response.status_code == 403:
            raise APIError("Доступ запрещен: недостаточно прав")

        elif response.status_code == 404:
            raise APIError("Ресурс не найден")

        elif response.status_code == 429:
            retry_after = response.headers.get('Retry-After', '60')
          raise APIError(f"Превышен лимит запросов. Повторите через {retry_
after} сек.")

        elif response.status_code >= 500:
            raise APIError("Внутренняя ошибка сервера")
```

```
        else:
            # Пытаемся извлечь сообщение об ошибке из JSON
            try:
                error_data = response.json()
                error_msg = error_data.get('message', 'Неизвестная ошибка')
                raise APIError(f"Ошибка API: {error_msg}")
            except:
                raise APIError(f"HTTP ошибка: {response.status_code}")

    except requests.exceptions.JSONDecodeError:
        raise APIError("Некорректный JSON в ответе API")
class APIError(Exception):
    """Исключение для ошибок API"""
    Pass
```

Работа с API требует понимания их документации, правильной обработки ошибок и соблюдения ограничений. Эти навыки критически важны для создания надежных приложений, интегрированных с внешними сервисами.

11.4. Базы данных: SQLite в Python

База данных (database) — это организованная коллекция структурированной информации, которая обычно хранится в компьютерной системе. Для Python-программ SQLite представляет идеальный выбор: это легковесная, встроенная база данных, не требующая отдельного сервера и идеально подходящая для изучения основ работы с данными.

Введение в SQLite и модуль sqlite3

SQLite — это встроенная реляционная СУБД (система управления базами данных), которая хранит всю базу данных в одном файле. Python включает модуль `sqlite3` в стандартную библиотеку, что делает работу с базами данных доступной без дополнительных установок.

```
import sqlite3
import os
from datetime import datetime
def create_database():
    """Создание базы данных и основных таблиц"""
    # Подключение к базе данных (файл создастся автоматически)
    conn = sqlite3.connect('library.db')
    cursor = conn.cursor()

    # Создание таблицы книг
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS books (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT NOT NULL,
```

```
        author TEXT NOT NULL,
        isbn TEXT UNIQUE,
        publication_year INTEGER,
        genre TEXT,
        available BOOLEAN DEFAULT 1,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
''')

# Создание таблицы читателей
cursor.execute('''
    CREATE TABLE IF NOT EXISTS readers (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        email TEXT UNIQUE NOT NULL,
        phone TEXT,
        registration_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
''')

# Создание таблицы выдач книг
cursor.execute('''
    CREATE TABLE IF NOT EXISTS loans (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        book_id INTEGER NOT NULL,
        reader_id INTEGER NOT NULL,
        loan_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        return_date TIMESTAMP,
        returned BOOLEAN DEFAULT 0,
        FOREIGN KEY (book_id) REFERENCES books (id),
        FOREIGN KEY (reader_id) REFERENCES readers (id)
    )
''')

# Сохранение изменений и закрытие соединения
conn.commit()
conn.close()
print("База данных создана успешно")
```

Этот код демонстрирует создание реляционной базы данных с тремя связанными таблицами, включая внешние ключи (foreign keys) для обеспечения целостности данных.

CRUD-операции: основа работы с данными

CRUD — это акроним от `Create` (создать), `Read` (прочитать), `Update` (обновить), `Delete` (удалить). Эти четыре операции составляют основу любой работы с базами данных.

Create: добавление данных

```
class LibraryDatabase:
    """Класс для работы с базой данных библиотеки"""

    def __init__(self, db_path='library.db'):
        self.db_path = db_path

    def get_connection(self):
        """Получение соединения с базой данных"""
        conn = sqlite3.connect(self.db_path)
        # Настраиваем возврат результатов как словари
        conn.row_factory = sqlite3.Row
        return conn

    def add_book(self, title, author, isbn=None, year=None, genre=None):
        """Добавление новой книги в базу данных"""
        conn = self.get_connection()
        cursor = conn.cursor()

        try:
            cursor.execute('''
             INSERT INTO books (title, author, isbn, publication_year, genre)
                VALUES (?, ?, ?, ?, ?)
            ''', (title, author, isbn, year, genre))

            book_id = cursor.lastrowid
            conn.commit()
            print(f"Книга добавлена с ID: {book_id}")
            return book_id

        except sqlite3.IntegrityError as e:
            print(f"Ошибка при добавлении книги: {e}")
            return None
        finally:
            conn.close()

    def add_reader(self, name, email, phone=None):
        """Добавление нового читателя"""
        conn = self.get_connection()
        cursor = conn.cursor()

        try:
            cursor.execute('''
                INSERT INTO readers (name, email, phone)
                VALUES (?, ?, ?)
            ''', (name, email, phone))

            reader_id = cursor.lastrowid
            conn.commit()
```

```
            print(f"Читатель зарегистрирован с ID: {reader_id}")
            return reader_id

        except sqlite3.IntegrityError as e:
            if 'email' in str(e):
                print("Пользователь с таким email уже существует")
            else:
                print(f"Ошибка при регистрации: {e}")
            return None
        finally:
            conn.close()
```

Использование параметризованных запросов (символ ?) защищает от SQL-инъекций — одной из самых распространенных уязвимостей веб-приложений.

Read: чтение и поиск данных

```
def get_books(self, search_term=None, available_only=False):
    """Получение списка книг с возможностью поиска"""
    conn = self.get_connection()
    cursor = conn.cursor()

    query = "SELECT * FROM books WHERE 1=1"
    params = []

    if search_term:
        query += " AND (title LIKE ? OR author LIKE ?)"
        search_pattern = f"%{search_term}%"
        params.extend([search_pattern, search_pattern])

    if available_only:
        query += " AND available = 1"

    query += " ORDER BY title"

    try:
        cursor.execute(query, params)
        books = cursor.fetchall()
        return [dict(book) for book in books]
    finally:
        conn.close()
def get_book_details(self, book_id):
    """Получение подробной информации о книге"""
    conn = self.get_connection()
    cursor = conn.cursor()

    try:
        # Основная информация о книге
```

```
        cursor.execute("SELECT * FROM books WHERE id = ?", (book_id,))
        book = cursor.fetchone()

        if not book:
            return None

        # Информация о текущих займах
        cursor.execute('''
            SELECT r.name, r.email, l.loan_date
            FROM loans l
            JOIN readers r ON l.reader_id = r.id
            WHERE l.book_id = ? AND l.returned = 0
        ''', (book_id,))

        current_loans = cursor.fetchall()

        book_dict = dict(book)
        book_dict['current_loans'] = [dict(loan) for loan in current_loans]

        return book_dict
    finally:
        conn.close()
```

Update: обновление данных

```
def update_book(self, book_id, **kwargs):
    """Обновление информации о книге"""
    if not kwargs:
        print("Нет данных для обновления")
        return False
    # Разрешенные поля для обновления
   allowed_fields = ['title', 'author', 'isbn', 'publication_year', 'genre',
'available']
    updates = {k: v for k, v in kwargs.items() if k in allowed_fields}

    if not updates:
        print("Нет допустимых полей для обновления")
        return False

    conn = self.get_connection()
    cursor = conn.cursor()

    try:
        # Строим запрос динамически
       set_clause = ", ".join([f"{field} = ?" for field in updates.keys()])
        query = f"UPDATE books SET {set_clause} WHERE id = ?"

        params = list(updates.values()) + [book_id]

        cursor.execute(query, params)

        if cursor.rowcount > 0:
            conn.commit()
```

```
            print(f"Книга с ID {book_id} обновлена")
            return True
        else:
            print(f"Книга с ID {book_id} не найдена")
            return False

    except sqlite3.Error as e:
        print(f"Ошибка при обновлении: {e}")
        return False
    finally:
        conn.close()
def loan_book(self, book_id, reader_id):
    """Выдача книги читателю"""
    conn = self.get_connection()
    cursor = conn.cursor()

    try:
        # Проверяем доступность книги
       cursor.execute("SELECT available FROM books WHERE id = ?", (book_id,))
        book = cursor.fetchone()

        if not book:
            print("Книга не найдена")
            return False

        if not book['available']:
            print("Книга уже выдана")
            return False

        # Начинаем транзакцию
        cursor.execute("BEGIN TRANSACTION")

        # Добавляем запись о выдаче
        cursor.execute('''
            INSERT INTO loans (book_id, reader_id)
            VALUES (?, ?)
        ''', (book_id, reader_id))

        # Помечаем книгу как недоступную
        cursor.execute('''
            UPDATE books SET available = 0 WHERE id = ?
        ''', (book_id,))

        conn.commit()
        print("Книга успешно выдана")
        return True

    except sqlite3.Error as e:
        conn.rollback()  # Откатываем транзакцию при ошибке
        print(f"Ошибка при выдаче книги: {e}")
        return False
```

```
    finally:
        conn.close()
```

Delete: удаление данных

```
def delete_book(self, book_id):
    """Удаление книги из базы данных"""
    conn = self.get_connection()
    cursor = conn.cursor()

    try:
        # Проверяем, есть ли активные займы
        cursor.execute('''
            SELECT COUNT(*) FROM loans
            WHERE book_id = ? AND returned = 0
        ''', (book_id,))

        active_loans = cursor.fetchone()[0]

        if active_loans > 0:
            print("Нельзя удалить книгу с активными займами")
            return False

        # Удаляем книгу
        cursor.execute("DELETE FROM books WHERE id = ?", (book_id,))

        if cursor.rowcount > 0:
            conn.commit()
            print(f"Книга с ID {book_id} удалена")
            return True
        else:
            print(f"Книга с ID {book_id} не найдена")
            return False

    except sqlite3.Error as e:
        print(f"Ошибка при удалении: {e}")
        return False
    finally:
        conn.close()
def return_book(self, loan_id):
    """Возврат книги в библиотеку"""
    conn = self.get_connection()
    cursor = conn.cursor()

    try:
        cursor.execute("BEGIN TRANSACTION")

        # Получаем информацию о займе
        cursor.execute('''
            SELECT book_id FROM loans
            WHERE id = ? AND returned = 0
```

```
        ''', (loan_id,))

        loan = cursor.fetchone()

        if not loan:
            print("Активный заём не найден")
            return False

        # Обновляем запись о займе
        cursor.execute('''
            UPDATE loans
            SET returned = 1, return_date = CURRENT_TIMESTAMP
            WHERE id = ?
        ''', (loan_id,))

        # Помечаем книгу как доступную
        cursor.execute('''
            UPDATE books SET available = 1 WHERE id = ?
        ''', (loan['book_id'],))

        conn.commit()
        print("Книга успешно возвращена")
        return True

    except sqlite3.Error as e:
        conn.rollback()
        print(f"Ошибка при возврате книги: {e}")
        return False
    finally:
        conn.close()
```

Основы ORM: упрощение работы с базами данных

ORM (Object-Relational Mapping) — это техника программирования, которая позволяет работать с базой данных как с объектами, скрывая SQL-запросы за высокоуровневым интерфейсом.

```
class Book:
    """Модель книги с ORM-подобным интерфейсом"""

     def __init__(self, title, author, isbn=None, year=None, genre=None,
book_id=None):
        self.id = book_id
        self.title = title
        self.author = author
        self.isbn = isbn
        self.publication_year = year
        self.genre = genre
        self.available = True

    @classmethod
    def create(cls, title, author, **kwargs):
```

```
        """Создание новой книги в базе данных"""
        db = LibraryDatabase()
        book_id = db.add_book(title, author, **kwargs)

        if book_id:
            book = cls(title, author, book_id=book_id, **kwargs)
            return book
        return None

    @classmethod
    def get_by_id(cls, book_id):
        """Получение книги по ID"""
        db = LibraryDatabase()
        conn = db.get_connection()
        cursor = conn.cursor()

        try:
            cursor.execute("SELECT * FROM books WHERE id = ?", (book_id,))
            row = cursor.fetchone()

            if row:
                return cls(
                    title=row['title'],
                    author=row['author'],
                    isbn=row['isbn'],
                    year=row['publication_year'],
                    genre=row['genre'],
                    book_id=row['id']
                )
            return None
        finally:
            conn.close()

    @classmethod
    def search(cls, search_term):
        """Поиск книг по названию или автору"""
        db = LibraryDatabase()
        books_data = db.get_books(search_term)

        return [
            cls(
                title=book['title'],
                author=book['author'],
                isbn=book['isbn'],
                year=book['publication_year'],
                genre=book['genre'],
                book_id=book['id']
            )
```

```
            for book in books_data
        ]

    def save(self):
        """Сохранение изменений в базе данных"""
        if self.id:
            # Обновляем существующую запись
            db = LibraryDatabase()
            return db.update_book(
                self.id,
                title=self.title,
                author=self.author,
                isbn=self.isbn,
                publication_year=self.publication_year,
                genre=self.genre
            )
        else:
            # Создаем новую запись
            new_book = self.create(
                self.title, self.author,
               isbn=self.isbn, year=self.publication_year, genre=self.genre
            )
            if new_book:
                self.id = new_book.id
                return True
            return False

    def __str__(self):
          return f"{self.title} - {self.author} ({self.publication_year or
'Год неизвестен'})"
    def __repr__(self):
          return f"Book(id={self.id}, title='{self.title}', author='{self.
author}')"
```

Этот подход к работе с базой данных делает код более читаемым и объектно-ориентированным, скрывая сложности SQL за понятными методами классов.

SQLite и модуль `sqlite3` предоставляют отличную основу для понимания принципов работы с базами данных. Освоив эти концепции, вы сможете легко перейти к более мощным СУБД, таким как PostgreSQL или MySQL, сохранив при этом понимание фундаментальных принципов работы с реляционными данными.

11.5. Регулярные выражения: мощный инструмент работы с текстом

Регулярные выражения (regular expressions) представляют собой формальный язык для описания шаблонов в тексте. Это один из самых мощных инструментов программиста для поиска, извлечения и обработки текстовых данных. Если строковые методы Python позволяют работать с точными совпадениями, то регулярные выражения дают возможность описать класс строк, соответствующих определенному шаблону.

Представьте, что вам нужно найти в тексте все email-адреса, телефонные номера или даты. Использование обычных строковых методов потребует написания громоздкого кода с множественными проверками. Регулярные выражения позволяют решить такую задачу элегантно — одним шаблоном, описывающим искомый паттерн.

Модуль re: основы работы с регулярными выражениями

Python предоставляет модуль `re` для работы с регулярными выражениями. Этот модуль содержит функции для поиска, замены и разбиения строк на основе шаблонов:

```
import re
# Простейший пример поиска
text = "Python is awesome. Python is powerful."
pattern = "Python"
matches = re.findall(pattern, text)
print(matches)  # ['Python', 'Python']
Основные функции модуля re:
import re
text = "Contact: john@example.com or call 555-1234"
# search() - поиск первого совпадения
match = re.search(r"\w+@\w+\.\w+", text)
if match:
    print("Email найден:", match.group())  # Email найден: john@example.com
# findall() - поиск всех совпадений
emails = re.findall(r"\w+@\w+\.\w+", text)
print("Все email:", emails)  # Все email: ['john@example.com']
# sub() - замена по шаблону
censored = re.sub(r"\d{3}-\d{4}", "XXX-XXXX", text)
print("Скрытый номер:", censored)  # Contact: john@example.com or call XXX-XXXX
```

Функция `re.search()` возвращает объект `Match` при успешном поиске или `None` при отсутствии совпадений. Функция `re.findall()` возвращает список всех найденных совпадений. Функция `re.sub()` выполняет замену всех совпадений с шаблоном на указанную строку.

Метасимволы: язык шаблонов

Сила регулярных выражений заключается в метасимволах — специальных символах, имеющих особое значение в шаблонах. Каждый метасимвол представляет класс символов или определяет правила поиска.

Основные метасимволы для поиска символов:

```
import re
text = "The price is $25.99 and the discount is 15%"
# . - любой символ, кроме переноса строки
print(re.findall(r".", "abc"))  # ['a', 'b', 'c']
# \d - любая цифра (эквивалент [0-9])
print(re.findall(r"\d", text))  # ['2', '5', '9', '9', '1', '5']
# \w - буква, цифра или подчеркивание
```

```
print(re.findall(r"\w+", "hello_world123"))  # ['hello_world123']
# \s – любой пробельный символ
print(re.findall(r"\s", text))  # [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Квантификаторы определяют количество повторений:
import re
# + – одно или более повторений
print(re.findall(r"\d+", "Price: $25.99"))  # ['25', '99']
# * – ноль или более повторений
print(re.findall(r"\w*", "hello world"))  # ['hello', '', 'world', '']
# ? – ноль или одно повторение
print(re.findall(r"colou?r", "color colour"))  # ['color', 'colour']
# {n,m} – от n до m повторений
print(re.findall(r"\d{3,4}", "123 4567 89"))  # ['123', '4567']
```

Символы позиции и границ:

```
import re
text = "Python programming"
# ^ – начало строки
print(re.search(r"^Python", text) is not None)  # True
print(re.search(r"^programming", text) is not None)  # False
# $ – конец строки
print(re.search(r"programming$", text) is not None)  # True
# \b – граница слова
print(re.findall(r"\bpro\w*", text))  # ['programming']
```

Символьные классы позволяют определить множество допустимых символов:

```
import re
# [abc] – любой из символов a, b или c
print(re.findall(r"[aeiou]", "hello"))  # ['e', 'o']
# [a-z] – любой символ в диапазоне от a до z
print(re.findall(r"[a-z]+", "Hello World"))  # ['ello', 'orld']
# [^abc] – любой символ, кроме a, b и c
print(re.findall(r"[^aeiou]", "hello"))  # ['h', 'l', 'l']
```

Группы: извлечение и структурирование данных

Группы в регулярных выражениях позволяют выделить части найденного совпадения для последующего использования. Группы создаются с помощью круглых скобок и предоставляют мощный механизм для структурирования извлекаемых данных.

Простые группы:

```
import re
# Извлечение имени и домена из email
email_pattern = r"(\w+)@(\w+\.\w+)"
text = "Contacts: alice@company.com, bob@test.org"
matches = re.findall(email_pattern, text)
for match in matches:
    username, domain = match
    print(f"Пользователь: {username}, домен: {domain}")
```

```
# Пользователь: alice, домен: company.com
# Пользователь: bob, домен: test.org
```

Именованные группы делают код более читаемым:

```
import re
# Использование именованных групп для разбора даты
date_pattern = r"(?P<day>\d{1,2})/(?P<month>\d{1,2})/(?P<year>\d{4})"
text = "Встреча назначена на 15/03/2024"
match = re.search(date_pattern, text)
if match:
    print("День:", match.group('day'))      # День: 15
    print("Месяц:", match.group('month'))   # Месяц: 03
    print("Год:", match.group('year'))      # Год: 2024
    print("Полная дата:", match.group(0))  # Полная дата: 15/03/2024
```

Группы также позволяют использовать альтернативы и опциональные части:

```
import re
# Поиск различных форматов телефонов
phone_pattern = r"(\+7|8)?[\s-]?(\d{3})[\s-]?(\d{3})[\s-]?(\d{2})
[\s-]?(\d{2})"
phones = [
    "+7 903 123 45 67",
    "8-903-123-45-67",
    "903 123 45 67",
    "9031234567"
]
for phone in phones:
    match = re.search(phone_pattern, phone)
    if match:
        code, num1, num2, num3, num4 = match.groups()
        formatted = f"{code or '8'} {num1} {num2} {num3} {num4}"
        print(f"Исходный: {phone} → Форматированный: {formatted}")
```

Практические примеры: решение реальных задач

Регулярные выражения особенно полезны при обработке неструктурированных текстовых данных. Рассмотрим несколько практических сценариев.

Валидация и извлечение данных:

```
import re
def validate_email(email):
    """Проверка корректности email-адреса"""
    pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return re.match(pattern, email) is not None
def extract_numbers(text):
    """Извлечение всех чисел из текста"""
    # Учитываем целые числа, десятичные дроби и отрицательные числа
    pattern = r"-?\d+(?:\.\d+)?"
    return [float(match) for match in re.findall(pattern, text)]
```

```
# Тестирование функций
emails = ["user@domain.com", "invalid.email", "test@test.co.uk"]
for email in emails:
    status = "корректный" if validate_email(email) else "некорректный"
    print(f"{email}: {status}")
text = "Температура -5.5°C, влажность 78%, давление 1013.25 гПа"
numbers = extract_numbers(text)
print(f"Найденные числа: {numbers}")  # [-5.5, 78, 1013.25]
```

Обработка логов и структурированных данных:

```
import re
from datetime import datetime
def parse_log_entry(log_line):
    """Разбор строки лога веб-сервера"""
    pattern = r'(\S+) - - \[([^\]]+)\] "(\w+) ([^"]*)" (\d+) (\d+)'
    match = re.match(pattern, log_line)

    if match:
        ip, timestamp, method, path, status, size = match.groups()
        return {
            'ip': ip,
            'timestamp': timestamp,
            'method': method,
            'path': path,
            'status': int(status),
            'size': int(size)
        }
    return None
# Пример строки лога
log_line = '192.168.1.100 - - [01/Jan/2024:12:00:00 +0000] "GET /index.html"
200 1234'
parsed = parse_log_entry(log_line)
if parsed:
    print(f"IP: {parsed['ip']}")
    print(f"Метод: {parsed['method']}")
    print(f"Статус: {parsed['status']}")
    print(f"Размер ответа: {parsed['size']} байт")
```

Очистка и нормализация текста:

```
import re
def clean_text(text):
    """Очистка текста от лишних символов и нормализация пробелов"""
    # Удаление HTML-тегов
    text = re.sub(r'<[^>]+>', '', text)

    # Замена множественных пробелов на одинарные
    text = re.sub(r'\s+', ' ', text)

    # Удаление пробелов в начале и конце
    text = text.strip()
```

```
    return text
def extract_hashtags(text):
    """Извлечение хештегов из текста"""
    return re.findall(r'#\w+', text)
# Пример использования
html_text = "<p>Отличная   статья о   <b>Python</b>! #python #programming</p>"
clean = clean_text(html_text)
hashtags = extract_hashtags(clean)
print(f"Очищенный текст: {clean}")
print(f"Хештеги: {hashtags}")
```

Работа с регулярными выражениями требует практики и внимательности. Важно помнить, что сложные шаблоны могут быть трудными для понимания и сопровождения. В таких случаях стоит разбивать сложный шаблон на несколько простых или добавлять подробные комментарии.

Регулярные выражения — это мощный инструмент, который значительно упрощает обработку текстовых данных. Освоение этой технологии открывает новые возможности для анализа неструктурированной информации, валидации пользовательского ввода и автоматизации текстовых операций. В следующих главах мы будем активно использовать регулярные выражения при работе с веб-данными и парсинге различных форматов.

Глава 12. Алгоритмы и структуры данных

12.1. Сложность алгоритмов: большое O

Представьте себе две дороги от дома до работы. Первая — прямая магистраль длиной 10 километров, вторая — живописный маршрут через город протяженностью 25 километров. В обычный день разница во времени может быть незначительной, но что происходит в час пик, когда на дорогах появляются тысячи автомобилей? Короткая дорога остается относительно быстрой, а длинная превращается в многочасовую пробку.

Аналогично работают алгоритмы. Два разных подхода к решению задачи могут показывать схожую производительность на небольших данных, но кардинально различаться при обработке миллионов элементов. Именно поэтому программисты изучают сложность алгоритмов — чтобы предсказать поведение программы при росте объема данных.

Временная сложность: как растет время выполнения

Временная сложность (time complexity) описывает зависимость времени выполнения алгоритма от размера входных данных. Это не точное время в секундах — оно зависит от мощности компьютера, языка программирования и множества других факторов. Вместо этого мы анализируем, как быстро растет время выполнения при увеличении размера данных.

Рассмотрим простую задачу поиска максимального элемента в списке:

```
def find_max_simple(numbers):
    """Простой поиск максимального элемента"""
    max_value = numbers[0]
    for number in numbers:
        if number > max_value:
            max_value = number
    return max_value
# Тестируем с разными размерами данных
small_list = [1, 5, 3, 9, 2]
large_list = list(range(100000))
print(find_max_simple(small_list))  # 9
```

В этом алгоритме мы просматриваем каждый элемент списка ровно один раз. Если в списке 5 элементов — делаем 5 сравнений, если 100 000 элементов — делаем 100 000 сравнений. Время выполнения растет линейно пропорционально размеру данных.

Сравним с менее эффективным подходом:

```
def find_max_nested(numbers):
    """Неэффективный поиск максимального элемента"""
    result = []
    for i in range(len(numbers)):
        is_maximum = True
```

```
        for j in range(len(numbers)):
            if numbers[j] > numbers[i]:
                is_maximum = False
                break
        if is_maximum:
            result.append(numbers[i])
    return max(result)
# Этот алгоритм работает значительно медленнее на больших данных
```

Здесь для каждого элемента мы проверяем все остальные элементы. При 5 элементах — 25 операций, при 100 элементах — 10 000 операций, при 1000 элементах — миллион операций. Время выполнения растет квадратично.

Нотация большого O

Нотация большого O (Big O notation) — это математический способ описания скорости роста функций. Она показывает верхнюю границу роста времени выполнения алгоритма в худшем случае.

Основные классы сложности, с которыми мы будем работать:

O(1) — константная сложность. Время выполнения не зависит от размера данных:

```
def get_first_element(data_list):
    """Получение первого элемента - всегда одна операция"""
    if len(data_list) > 0:
        return data_list[0]
    return None
def dict_lookup(dictionary, key):
    """Поиск в словаре - примерно константное время"""
    return dictionary.get(key)
# Размер данных не влияет на скорость этих операций
small_dict = {"a": 1, "b": 2}
large_dict = {f"key_{i}": i for i in range(100000)}
# Поиск займет примерно одинаковое время
print(dict_lookup(small_dict, "a"))
print(dict_lookup(large_dict, "key_50000"))
```

O(log n) — логарифмическая сложность. Характерна для алгоритмов, которые на каждом шаге уменьшают размер задачи вдвое:

```
def binary_search_recursive(sorted_list, target, left=0, right=None):
    """Бинарный поиск с рекурсией"""
    if right is None:
        right = len(sorted_list) - 1

    if left > right:
        return -1  # Элемент не найден

    mid = (left + right) // 2

    if sorted_list[mid] == target:
        return mid
```

```
    elif sorted_list[mid] > target:
        return binary_search_recursive(sorted_list, target, left, mid - 1)
    else:
        return binary_search_recursive(sorted_list, target, mid + 1, right)
# Для списка из миллиона элементов понадобится максимум 20 операций
data = list(range(1000000))
print(binary_search_recursive(data, 750000))
```

O(n) — линейная сложность. Время выполнения пропорционально размеру данных:

```
def linear_search(data_list, target):
    """Линейный поиск элемента"""
    for i, element in enumerate(data_list):
        if element == target:
            return i
    return -1
def calculate_sum(numbers):
    """Вычисление суммы элементов"""
    total = 0
    for number in numbers:
        total += number
    return total
# Время выполнения прямо пропорционально количеству элементов
```

O(n^2) — квадратичная сложность. Характерна для вложенных циклов:

```
def bubble_sort_basic(arr):
    """Простая сортировка пузырьком"""
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
def find_duplicates_naive(data_list):
    """Поиск дублирующихся элементов неэффективным способом"""
    duplicates = []
    for i in range(len(data_list)):
        for j in range(i + 1, len(data_list)):
          if data_list[i] == data_list[j] and data_list[i] not in duplicates:
                duplicates.append(data_list[i])
    return duplicates
# На больших данных эти алгоритмы работают очень медленно
test_data = [3, 1, 4, 1, 5, 9, 2, 6, 5]
print(find_duplicates_naive(test_data))  # [1, 5]
```

Пространственная сложность

Пространственная сложность (space complexity) описывает количество дополнительной памяти, которое требует алгоритм для своей работы. Это особенно важно при работе с большими объемами данных.

Рассмотрим разные подходы к переворачиванию списка:

```
def reverse_new_list(original_list):
    """Создание нового списка - O(n) по памяти"""
    return original_list[::-1]
def reverse_in_place(data_list):
    """Изменение исходного списка - O(1) по памяти"""
    left = 0
    right = len(data_list) - 1

    while left < right:
        # Меняем элементы местами
      data_list[left], data_list[right] = data_list[right], data_list[left]
        left += 1
        right -= 1

    return data_list
# Первый способ создает новый список, второй изменяет существующий
original = [1, 2, 3, 4, 5]
copy_for_test = original.copy()
reversed_new = reverse_new_list(original)
print(f"Оригинальный: {original}")
print(f"Новый обращенный: {reversed_new}")
reverse_in_place(copy_for_test)
print(f"Обращенный на месте: {copy_for_test}")
```

Для демонстрации различий в пространственной сложности создадим алгоритм поиска всех подстрок:

```
def find_substrings_memory_heavy(text):
    """Генерация всех подстрок - высокое потребление памяти"""
    substrings = []
    n = len(text)

    for i in range(n):
        for j in range(i + 1, n + 1):
            substrings.append(text[i:j])

    return substrings
def find_substrings_generator(text):
    """Генератор подстрок - низкое потребление памяти"""
    n = len(text)

    for i in range(n):
        for j in range(i + 1, n + 1):
            yield text[i:j]
# Первый способ создает список всех подстрок в памяти
# Второй генерирует их по одной при необходимости
text_sample = "Python"
all_substrings = find_substrings_memory_heavy(text_sample)
print(f"Всего подстрок: {len(all_substrings)}")
print(f"Первые 10: {all_substrings[:10]}")
```

```
# Генератор потребляет значительно меньше памяти
substring_generator = find_substrings_generator(text_sample)
print("Первые 5 подстрок из генератора:")
for i, substring in enumerate(substring_generator):
    if i >= 5:
        break
    print(f"  {substring}")
```

Анализ реальных задач

Применим полученные знания к анализу практических алгоритмов. Начнем с задачи подсчета частоты слов в тексте — классической задачи анализа данных.

```
def count_words_naive(text):
    """Наивный подсчет слов - O(n²) по времени"""
    words = text.lower().split()
    word_counts = []

    for word in words:
        # Ищем, встречалось ли уже это слово
        found = False
        for i, (existing_word, count) in enumerate(word_counts):
            if existing_word == word:
                word_counts[i] = (word, count + 1)
                found = True
                break

        if not found:
            word_counts.append((word, 1))

    return word_counts
def count_words_efficient(text):
    """Эффективный подсчет с использованием словаря - O(n)"""
    words = text.lower().split()
    word_counts = {}

    for word in words:
        word_counts[word] = word_counts.get(word, 0) + 1

    return word_counts
# Тестируем на примере текста
sample_text = """
Python is a powerful programming language.
Python is easy to learn and Python is widely used.
Programming in Python is fun and programming is rewarding.
"""
# Измеряем производительность (концептуально)
print("Наивный алгоритм:")
naive_result = count_words_naive(sample_text)
for word, count in naive_result[:5]:  # Показываем первые 5
    print(f"  {word}: {count}")
```

```
print("\nЭффективный алгоритм:")
efficient_result = count_words_efficient(sample_text)
for word, count in list(efficient_result.items())[:5]:
    print(f"  {word}: {count}")
```

Рассмотрим еще один пример — поиск пересечения двух списков:

```
def find_intersection_nested(list1, list2):
    """Поиск пересечения через вложенные циклы - O(n*m)"""
    intersection = []

    for element1 in list1:
        for element2 in list2:
            if element1 == element2 and element1 not in intersection:
                intersection.append(element1)

    return intersection
def find_intersection_sets(list1, list2):
    """Поиск пересечения через множества - O(n + m)"""
    set1 = set(list1)
    set2 = set(list2)
    return list(set1.intersection(set2))
def find_intersection_dict(list1, list2):
    """Поиск пересечения через словарь - O(n + m)"""
    # Создаем словарь для быстрого поиска
    dict1 = {element: True for element in list1}
    intersection = []

    for element in list2:
        if element in dict1 and element not in intersection:
            intersection.append(element)

    return intersection
# Сравниваем подходы
first_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
second_list = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
print("Пересечение (вложенные циклы):", find_intersection_nested(first_
list, second_list))
print("Пересечение (множества):", find_intersection_sets(first_list,
second_list))
print("Пересечение (словарь):", find_intersection_dict(first_list, second_
list))
```

Амортизированная сложность

Некоторые операции имеют различную сложность в разных случаях. Например, добавление элемента в динамический список (list) в Python обычно выполняется за `O(1)`, но иногда требует `O(n)` времени для расширения внутреннего массива:

```
import time
def measure_list_append():
    """Демонстрация амортизированной сложности добавления в список"""
```

```
    test_list = []
    times = []

    for i in range(1000):
        start_time = time.time()
        test_list.append(i)
        end_time = time.time()

        times.append(end_time - start_time)

        # Выводим время для некоторых операций
        if i in [10, 50, 100, 500, 999]:
            avg_time = sum(times) / len(times)
            print(f"После {i+1} добавлений, среднее время: {avg_time:.8f}")
# Большинство операций быстрые, но периодически происходят медленные
measure_list_append()
```

Практические рекомендации

При выборе алгоритма учитывайте не только асимптотическую сложность, но и константные факторы, размер реальных данных и особенности применения:

```
def choose_sorting_algorithm(data, data_size_estimate):
    """Выбор алгоритма сортировки в зависимости от условий"""

    if data_size_estimate < 50:
        # Для малых данных простые алгоритмы могут быть эффективнее
        return "insertion_sort"  # O(n²), но с малыми константами
    elif data_size_estimate < 1000:
        # Для средних данных подходят улучшенные алгоритмы
        return "merge_sort"      # O(n log n), стабильная
    else:
        # Для больших данных нужны самые эффективные алгоритмы
        return "timsort"         # Используется в Python по умолчанию
def analyze_algorithm_choice(algorithm_name, data_size):
    """Анализ выбора алгоритма"""
    complexity_info = {
        "insertion_sort": "O(n²) - хорош для малых данных",
        "merge_sort": "O(n log n) - стабильная сортировка",
        "timsort": "O(n log n) - адаптивная, используется в sorted()"
    }

    print(f"Для {data_size} элементов рекомендуется: {algorithm_name}")
    print(f"Сложность: {complexity_info[algorithm_name]}")
# Демонстрируем принцип выбора
test_sizes = [25, 500, 50000]
for size in test_sizes:
    chosen_algorithm = choose_sorting_algorithm([], size)
    analyze_algorithm_choice(chosen_algorithm, size)
    print()
```

Понимание сложности алгоритмов — это компас в мире программирования. Он помогает не потеряться среди множества возможных решений и выбрать наиболее подходящий путь к цели. В следующих разделах мы применим эти знания к изучению конкретных алгоритмов поиска и сортировки, увидим, как теория воплощается в практические решения.

12.2. Алгоритмы поиска: линейный и бинарный

Поиск — одна из фундаментальных задач в программировании. Представьте, что у вас есть коллекция данных, и вам необходимо найти конкретный элемент. Эффективность этого процесса напрямую влияет на производительность всего приложения. В этом разделе мы изучим два базовых алгоритма поиска: линейный и бинарный, проанализируем их сложность и поймем, когда каждый из них наиболее применим.

Линейный поиск: простота и универсальность

Линейный поиск (linear search) — самый интуитивный алгоритм поиска. Его принцип прост: мы последовательно проверяем каждый элемент коллекции, пока не найдем искомое значение или не исчерпаем все элементы. Этот метод работает с любыми данными, независимо от их упорядоченности.

Рассмотрим базовую реализацию линейного поиска:

```
def linear_search(arr, target):
    """
    Линейный поиск элемента в массиве.

    Args:
        arr: список для поиска
        target: искомый элемент

    Returns:
        индекс найденного элемента или -1, если элемент не найден
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
# Пример использования
numbers = [64, 34, 25, 12, 22, 11, 90]
result = linear_search(numbers, 22)
print(f"Элемент найден на позиции: {result}")  # Элемент найден на позиции: 4
# Поиск отсутствующего элемента
result = linear_search(numbers, 100)
print(f"Элемент найден на позиции: {result}")  # Элемент найден на позиции: -1
```

Временная сложность линейного поиска составляет `O(n)`, где `n` — количество элементов в коллекции. В худшем случае нам придется проверить все элементы. Пространственная сложность — `O(1)`, поскольку мы используем только фиксированное количество дополнительной памяти.

Линейный поиск можно адаптировать для различных задач. Например, найти все вхождения элемента:

```
def linear_search_all(arr, target):
    """Найти все позиции элемента в массиве."""
    positions = []
    for i, element in enumerate(arr):
        if element == target:
            positions.append(i)
    return positions
# Пример с повторяющимися элементами
data = [1, 3, 7, 3, 9, 3, 5]
all_positions = linear_search_all(data, 3)
print(f"Элемент 3 найден на позициях: {all_positions}")  # [1, 3, 5]
```

Или найти элемент, удовлетворяющий определенному условию:

```
def linear_search_condition(arr, condition):
    """Найти первый элемент, удовлетворяющий условию."""
    for i, element in enumerate(arr):
        if condition(element):
            return i, element
    return -1, None
# Поиск первого четного числа
numbers = [1, 3, 5, 8, 9, 12, 15]
index, value = linear_search_condition(numbers, lambda x: x % 2 == 0)
print(f"Первое четное число {value} на позиции {index}")  # Первое четное
число 8 на позиции 3
```

Бинарный поиск: эффективность через упорядоченность

Бинарный поиск (binary search) — значительно более эффективный алгоритм, но с важным ограничением: данные должны быть отсортированы. Принцип работы основан на стратегии «разделяй и властвуй»: мы сравниваем искомый элемент со средним элементом массива и исключаем половину элементов из дальнейшего рассмотрения.

Рассмотрим итеративную реализацию бинарного поиска:

```
def binary_search(arr, target):
    """
    Бинарный поиск в отсортированном массиве.

    Args:
        arr: отсортированный список для поиска
        target: искомый элемент

    Returns:
        индекс найденного элемента или -1, если элемент не найден
    """
    left, right = 0, len(arr) - 1

    while left <= right:
```

```
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1  # Искать в правой половине
        else:
            right = mid - 1  # Искать в левой половине

    return -1
# Пример использования
sorted_numbers = [11, 12, 22, 25, 34, 64, 90]
result = binary_search(sorted_numbers, 25)
print(f"Элемент найден на позиции: {result}")  # Элемент найден на пози-
ции: 3
```

Временная сложность бинарного поиска — `O(log n)`, что является значительным улучшением по сравнению с линейным поиском. Пространственная сложность также составляет `O(1)` для итеративной версии.

Рекурсивная реализация бинарного поиска более интуитивна для понимания принципа «разделяй и властвуй»:

```
def binary_search_recursive(arr, target, left=0, right=None):
    """Рекурсивная реализация бинарного поиска."""
    if right is None:
        right = len(arr) - 1

    if left > right:
        return -1  # Базовый случай: элемент не найден

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)
# Тестирование рекурсивной версии
result = binary_search_recursive(sorted_numbers, 64)
print(f"Рекурсивный поиск: элемент найден на позиции {result}")
```

Визуализация процесса поиска

Для лучшего понимания различий между алгоритмами создадим функции, которые показывают процесс поиска:

```
def linear_search_verbose(arr, target):
    """Линейный поиск с выводом промежуточных шагов."""
    print(f"Линейный поиск элемента {target}")
    comparisons = 0
```

```
    for i in range(len(arr)):
        comparisons += 1
        print(f"Шаг {comparisons}: проверяем arr[{i}] = {arr[i]}")

        if arr[i] == target:
           print(f"Элемент найден! Позиция: {i}, сравнений: {comparisons}")
            return i

    print(f"Элемент не найден. Всего сравнений: {comparisons}")
    return -1
def binary_search_verbose(arr, target):
    """Бинарный поиск с выводом промежуточных шагов."""
    print(f"Бинарный поиск элемента {target}")
    left, right = 0, len(arr) - 1
    comparisons = 0

    while left <= right:
        comparisons += 1
        mid = (left + right) // 2
        print(f"Шаг {comparisons}: проверяем arr[{mid}] = {arr[mid]} "
              f"(диапазон: [{left}:{right}])")

        if arr[mid] == target:
          print(f"Элемент найден! Позиция: {mid}, сравнений: {comparisons}")
            return mid
        elif arr[mid] < target:
            print(f"Ищем в правой половине")
            left = mid + 1
        else:
            print(f"Ищем в левой половине")
            right = mid - 1

    print(f"Элемент не найден. Всего сравнений: {comparisons}")
    return -1
```

Протестируем оба алгоритма на одном примере:

```
test_array = [10, 23, 45, 70, 80, 99, 110, 130, 150, 170]
target = 110
print("=" * 50)
linear_result = linear_search_verbose(test_array, target)
print("\n" + "=" * 50)
binary_result = binary_search_verbose(test_array, target)
```

Сравнительный анализ производительности

Для количественной оценки эффективности алгоритмов создадим функцию измерения времени выполнения:

```
import time
import random
```

```
def measure_search_performance(search_func, arr, target, iterations=1000):
    """Измерение времени выполнения поиска."""
    start_time = time.time()

    for _ in range(iterations):
        search_func(arr, target)

    end_time = time.time()
    return (end_time - start_time) / iterations
# Создание тестовых данных
sizes = [100, 1000, 10000, 100000]
for size in sizes:
    # Создаем отсортированный массив
    test_data = sorted(random.sample(range(size * 10), size))
    target = random.choice(test_data)

      linear_time = measure_search_performance(linear_search, test_data,
target)
      binary_time = measure_search_performance(binary_search, test_data,
target)

    print(f"Размер массива: {size}")
    print(f"Линейный поиск: {linear_time:.6f} сек")
    print(f"Бинарный поиск:  {binary_time:.6f} сек")
    print(f"Ускорение: {linear_time/binary_time:.1f}x")
    print("-" * 40)
```

Практическое применение и вариации

В реальных приложениях алгоритмы поиска часто адаптируются под конкретные задачи. Рассмотрим несколько практических вариаций.

Поиск диапазона значений в отсортированном массиве:

```
def find_range(arr, target):
    """Найти первое и последнее вхождение элемента."""
    def find_first():
        left, right = 0, len(arr) - 1
        result = -1

        while left <= right:
            mid = (left + right) // 2
            if arr[mid] == target:
                result = mid
                right = mid - 1  # Продолжаем поиск левее
            elif arr[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return result
```

```
    def find_last():
        left, right = 0, len(arr) - 1
        result = -1

        while left <= right:
            mid = (left + right) // 2
            if arr[mid] == target:
                result = mid
                left = mid + 1  # Продолжаем поиск правее
            elif arr[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return result

    first = find_first()
    if first == -1:
        return -1, -1

    last = find_last()
    return first, last
# Тестирование
data = [1, 2, 3, 3, 3, 3, 4, 5, 6]
first, last = find_range(data, 3)
print(f"Элемент 3 встречается с позиции {first} по {last}")  # 2 по 5
```

Поиск в поворачиваемом отсортированном массиве:

```
def search_rotated(arr, target):
    """
    Поиск в отсортированном массиве, который был повернут.
    Например: [4,5,6,7,0,1,2] - поворот массива [0,1,2,4,5,6,7]
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid

        # Определяем, какая половина отсортирована
        if arr[left] <= arr[mid]:  # Левая половина отсортирована
            if arr[left] <= target < arr[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:  # Правая половина отсортирована
            if arr[mid] < target <= arr[right]:
                left = mid + 1
            else:
                right = mid - 1
```

```
    return -1
# Пример использования
rotated = [4, 5, 6, 7, 0, 1, 2]
result = search_rotated(rotated, 0)
print(f"В поворачиваемом массиве элемент 0 на позиции: {result}")  # 4
```

Выбор алгоритма поиска

При выборе между линейным и бинарным поиском учитывайте следующие факторы:

Используйте линейный поиск, когда:

- данные не отсортированы и сортировка не оправдана;
- размер коллекции небольшой (меньше 100-1000 элементов);
- необходимо найти все вхождения элемента;
- работаете со связанными списками или потоками данных.

Используйте бинарный поиск, когда:

- данные отсортированы или можно их отсортировать;
- размер коллекции большой;
- выполняется много операций поиска;
- важна производительность поиска.

Важно понимать, что стоимость сортировки для бинарного поиска составляет `O(n log n)`. Если вы планируете выполнить только один поиск, то сортировка + бинарный поиск может быть менее эффективной, чем простой линейный поиск.

В следующем разделе мы изучим алгоритмы сортировки, которые являются необходимой предпосылкой для эффективного использования бинарного поиска и многих других алгоритмов.

12.3. Алгоритмы сортировки: от пузырька до быстрой

Сортировка данных — одна из фундаментальных задач в программировании. Способность расположить элементы в определенном порядке лежит в основе многих алгоритмов поиска, анализа данных и оптимизации. В этом разделе мы изучим пять классических алгоритмов сортировки, каждый из которых демонстрирует различные подходы к решению одной задачи.

Понимание принципов работы различных алгоритмов сортировки важно не только для практического применения, но и для развития алгоритмического мышления. Каждый алгоритм имеет свои преимущества и недостатки, что делает их подходящими для разных сценариев использования.

Сортировка пузырьком (Bubble Sort)

Сортировка пузырьком — это простейший алгоритм сортировки, который получил свое название из-за того, что большие элементы «всплывают» к концу массива подобно пузырькам воздуха в воде. Алгоритм работает путем многократного прохода

по списку, сравнивая соседние элементы и меняя их местами, если они расположены в неправильном порядке.

Основная идея заключается в том, что после каждого полного прохода самый большой элемент гарантированно оказывается в правильной позиции в конце массива. Таким образом, с каждой итерацией область сортировки уменьшается на один элемент.

```
def bubble_sort(arr):
    """
    Сортировка пузырьком - простой, но медленный алгоритм
    Временная сложность: O(n²)
    Пространственная сложность: O(1)
    """
    n = len(arr)

    # Проходим по всему массиву
    for i in range(n):
        # Флаг для оптимизации - отслеживаем, были ли обмены
        swapped = False

        # Последние i элементов уже отсортированы
        for j in range(0, n - i - 1):
            # Сравниваем соседние элементы
            if arr[j] > arr[j + 1]:
                # Меняем местами, если порядок неправильный
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # Если обменов не было, массив отсортирован
        if not swapped:
            break

    return arr
```

Давайте проследим работу алгоритма на примере:

```
# Пример работы пузырьковой сортировки
numbers = [64, 34, 25, 12, 22, 11, 90]
print("Исходный массив:", numbers)

# Создаем копию для демонстрации
demo_array = numbers.copy()
result = bubble_sort(demo_array)
print("Отсортированный массив:", result)
```

Хотя пузырьковая сортировка интуитивно понятна и проста в реализации, ее временная сложность $O(n^2)$ делает ее неэффективной для больших наборов данных. Однако алгоритм имеет важное преимущество — он стабилен, то есть сохраняет относительный порядок равных элементов.

Сортировка выбором (Selection Sort)

Сортировка выбором работает по принципу поиска минимального (или максимального) элемента в неотсортированной части массива и перемещения его в начало отсортированной части. Алгоритм разделяет входной список на две части: отсортированную подпоследовательность в начале и неотсортированную в конце.

На каждой итерации алгоритм находит наименьший элемент в неотсортированной части и обменивает его с первым элементом неотсортированной части, тем самым расширяя отсортированную область на один элемент:

```
def selection_sort(arr):
    """
    Сортировка выбором - выбираем минимальный элемент
    Временная сложность: O(n²)
    Пространственная сложность: O(1)
    """
    n = len(arr)

    # Проходим по всему массиву
    for i in range(n):
        # Предполагаем, что минимум находится в позиции i
        min_index = i

        # Ищем минимальный элемент в оставшейся части
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        # Меняем местами найденный минимум с первым элементом
        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr
```

Важной особенностью сортировки выбором является то, что количество обменов элементов минимально — не более n-1 обменов для массива из n элементов. Это может быть важно в ситуациях, когда операция обмена является дорогостоящей.

```
# Демонстрация сортировки выбором
data = [29, 10, 14, 37, 13]
print("До сортировки:", data)

selection_sort(data)
print("После сортировки:", data)
```

Сортировка вставками (Insertion Sort)

Сортировка вставками работает аналогично тому, как большинство людей сортируют игральные карты в руке. Алгоритм строит финальный отсортированный массив по одному элементу за раз, вставляя каждый новый элемент в правильную позицию среди уже отсортированных элементов.

Алгоритм эффективен для небольших наборов данных и обладает важным свойством — он адаптивен, то есть работает быстрее на частично отсортированных данных. Кроме того, он стабилен и работает «на месте» (in-place), не требуя дополнительной памяти:

```
def insertion_sort(arr):
    """
    Сортировка вставками - вставляем элементы в нужное место
    Временная сложность: O(n²) в худшем случае, O(n) в лучшем
    Пространственная сложность: O(1)
    """
    # Начинаем со второго элемента (индекс 1)
    for i in range(1, len(arr)):
        key = arr[i]  # Элемент, который нужно вставить
        j = i - 1     # Индекс последнего элемента в отсортированной части

        # Двигаем элементы больше key на одну позицию вперед
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Вставляем key в найденное место
        arr[j + 1] = key

    return arr
```

Одним из главных преимуществ сортировки вставками является ее производительность на маленьких массивах и почти отсортированных данных. Многие продвинутые алгоритмы сортировки используют вставки как вспомогательный метод для обработки небольших подмассивов.

```
# Пример работы сортировки вставками
cards = [5, 2, 4, 6, 1, 3]
print("Карты до сортировки:", cards)

insertion_sort(cards)
print("Карты после сортировки:", cards)
```

Сортировка слиянием (Merge Sort)

Сортировка слиянием представляет собой классический пример применения принципа «разделяй и властвуй» (divide and conquer). Алгоритм рекурсивно разделяет массив пополам до тех пор, пока не останутся массивы из одного элемента, а затем объединяет их обратно в отсортированном порядке.

Ключевой операцией является слияние двух отсортированных подмассивов в один отсортированный массив. Этот процесс выполняется за линейное время и является основой эффективности всего алгоритма:

```
def merge_sort(arr):
    """
    Сортировка слиянием - рекурсивный алгоритм "разделяй и властвуй"
    Временная сложность: O(n log n)
```

```
    Пространственная сложность: O(n)
    """
    if len(arr) <= 1:
        return arr

    # Разделяем массив пополам
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    # Рекурсивно сортируем обе части
    left = merge_sort(left)
    right = merge_sort(right)

    # Объединяем отсортированные части
    return merge(left, right)

def merge(left, right):
    """
    Функция слияния двух отсортированных массивов
    """
    result = []
    i, j = 0, 0

    # Сравниваем элементы и добавляем меньший в результат
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Добавляем оставшиеся элементы
    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

Сортировка слиянием гарантирует временную сложность `O(n log n)` в любом случае, что делает ее предсказуемой и надежной для больших наборов данных. Однако она требует дополнительной памяти, пропорциональной размеру входного массива.

```
# Демонстрация сортировки слиянием
large_data = [38, 27, 43, 3, 9, 82, 10]
print("Исходные данные:", large_data)

sorted_data = merge_sort(large_data)
print("После сортировки слиянием:", sorted_data)
```

Быстрая сортировка (Quick Sort)

Быстрая сортировка — это еще один алгоритм, основанный на принципе «разделяй и властвуй», но с другим подходом к разбиению. Алгоритм выбирает элемент в качестве опорного (pivot), разбивает массив на элементы меньше опорного и больше опорного, а затем рекурсивно сортирует получившиеся части.

Эффективность алгоритма сильно зависит от выбора опорного элемента. В худшем случае (когда опорный элемент всегда оказывается наименьшим или наибольшим) сложность составляет $O(n^2)$, но в среднем случае алгоритм показывает отличную производительность $O(n \log n)$:

```
def quick_sort(arr, low=0, high=None):
    """
    Быстрая сортировка - эффективный алгоритм "разделяй и властвуй"
    Временная сложность: O(n log n) в среднем, O(n²) в худшем случае
    Пространственная сложность: O(log n) для рекурсии
    """
    if high is None:
        high = len(arr) - 1

    if low < high:
        # Получаем индекс опорного элемента
        pi = partition(arr, low, high)

        # Рекурсивно сортируем элементы до и после опорного
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

    return arr

def partition(arr, low, high):
    """
    Функция разбиения - помещает опорный элемент в правильную позицию
    """
    # Выбираем последний элемент как опорный
    pivot = arr[high]

    # Индекс меньшего элемента
    i = low - 1

    for j in range(low, high):
        # Если текущий элемент меньше или равен опорному
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Помещаем опорный элемент в правильную позицию
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Быстрая сортировка работает «на месте» и показывает отличную производительность на практике. Однако она не является стабильной сортировкой и может деградировать до O(n²) на некоторых входных данных.

```
# Пример быстрой сортировки
quick_data = [10, 7, 8, 9, 1, 5]
print("До быстрой сортировки:", quick_data)

quick_sort(quick_data)
print("После быстрой сортировки:", quick_data)
```

Сравнение алгоритмов сортировки

Каждый алгоритм сортировки имеет свои характеристики, которые определяют области его применения. Понимание этих различий помогает выбрать подходящий алгоритм для конкретной задачи.

```
def compare_sorting_algorithms():
    """
    Функция для сравнения различных алгоритмов сортировки
    """
    import time
    import random

    # Создаем тестовые данные
    size = 1000
    test_data = [random.randint(1, 1000) for _ in range(size)]

    algorithms = {
        'Bubble Sort': bubble_sort,
        'Selection Sort': selection_sort,
        'Insertion Sort': insertion_sort,
        'Merge Sort': merge_sort,
        'Quick Sort': quick_sort
    }

    print(f"Сравнение алгоритмов на массиве из {size} элементов:")
    print("-" * 50)

    for name, func in algorithms.items():
        data_copy = test_data.copy()
        start_time = time.time()

        if name == 'Quick Sort':
            func(data_copy)
        else:
            func(data_copy)

        end_time = time.time()
        execution_time = end_time - start_time

        print(f"{name}: {execution_time:.4f} секунд")
```

Выбор алгоритма сортировки зависит от нескольких факторов: размера данных, требований к памяти, необходимости стабильности сортировки и характеристик входных данных. Для небольших массивов часто эффективна сортировка вставками, для больших данных предпочтительны `merge sort` или `quick sort`, а для случаев с ограниченной памятью — алгоритмы, работающие «на месте».

Понимание принципов работы этих базовых алгоритмов закладывает фундамент для изучения более сложных структур данных и алгоритмов, а также развивает способность анализировать и оценивать эффективность различных подходов к решению задач.

12.4. Стеки и очереди: LIFO и FIFO

Представьте стопку тарелок на кухне или очередь в магазине — эти простые житейские ситуации отражают фундаментальные принципы организации данных в программировании. **Стеки** и **очереди** являются линейными структурами данных, которые определяют строгие правила добавления и удаления элементов.

Стек (stack) работает по принципу LIFO — Last In, First Out (последний пришел — первый ушел). Как стопка тарелок: новую тарелку кладут сверху, и когда нужна тарелка, берут тоже верхнюю. **Очередь (queue)** следует принципу FIFO — First In, First Out (первый пришел — первый ушел), подобно очереди людей: кто пришел раньше, тот и обслуживается первым.

Стеки: последний пришел — первый ушел

Стек поддерживает две основные операции: **push** (положить элемент на верх) и **pop** (взять элемент с верха). В Python проще всего реализовать стек с помощью обычного списка, используя методы `append()` и `pop()`.

```
class Stack:
    """Реализация стека на основе списка Python"""

    def __init__(self):
        self._items = []

    def push(self, item):
        """Добавить элемент на верх стека"""
        self._items.append(item)

    def pop(self):
        """Удалить и вернуть верхний элемент"""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Посмотреть верхний элемент без удаления"""
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]
```

```
    def is_empty(self):
        """Проверить, пуст ли стек"""
        return len(self._items) == 0

    def size(self):
        """Получить количество элементов"""
        return len(self._items)
# Пример использования стека
stack = Stack()
stack.push("первый")
stack.push("второй")
stack.push("третий")
print(f"Размер стека: {stack.size()}")  # 3
print(f"Верхний элемент: {stack.peek()}")  # "третий"
while not stack.is_empty():
    print(f"Извлекаем: {stack.pop()}")
# Вывод:
# Извлекаем: третий
# Извлекаем: второй
# Извлекаем: первый
```

Стеки находят применение во многих алгоритмах. Например, при проверке правильности расстановки скобок в математических выражениях:

```
def is_balanced_parentheses(expression):
    """Проверка сбалансированности скобок"""
    stack = Stack()
    opening = "([{"
    closing = ")]}"
    pairs = {"(": ")", "[": "]", "{": "}"}

    for char in expression:
        if char in opening:
            stack.push(char)
        elif char in closing:
            if stack.is_empty():
                return False
            if pairs[stack.pop()] != char:
                return False

    return stack.is_empty()
# Тестирование
test_expressions = [
    "(())",           # True
    "([{}])",         # True
    "([)]",           # False
    "(((",            # False
])
for expr in test_expressions:
    result = is_balanced_parentheses(expr)
    print(f"'{expr}' → {result}")
```

Очереди: первый пришел — первый ушел

Очередь поддерживает операции **enqueue** (добавить элемент в конец) и **dequeue** (удалить элемент из начала). Хотя можно использовать список, операция удаления с начала списка (`pop(0)`) имеет временную сложность `O(n)`, что неэффективно.

```
class SimpleQueue:
    """Простая реализация очереди на основе списка"""

    def __init__(self):
        self._items = []

    def enqueue(self, item):
        """Добавить элемент в конец очереди"""
        self._items.append(item)

    def dequeue(self):
        """Удалить и вернуть первый элемент"""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)  # Неэффективно!

    def front(self):
        """Посмотреть первый элемент без удаления"""
        if self.is_empty():
            raise IndexError("front from empty queue")
        return self._items[0]

    def is_empty(self):
        return len(self._items) == 0

    def size(self):
        return len(self._items)
# Демонстрация работы очереди
queue = SimpleQueue()
queue.enqueue("Alice")
queue.enqueue("Bob")
queue.enqueue("Charlie")
print(f"Первый в очереди: {queue.front()}")  # Alice
while not queue.is_empty():
    print(f"Обслуживаем: {queue.dequeue()}")
# Вывод:
# Обслуживаем: Alice
# Обслуживаем: Bob
# Обслуживаем: Charlie
```

Эффективная реализация с помощью deque

Модуль collections предоставляет класс **deque** (double-ended queue — двухсторонняя очередь), оптимизированный для быстрых операций добавления и удаления с обоих концов.

```
from collections import deque
class EfficientQueue:
    """Эффективная реализация очереди с помощью deque"""

    def __init__(self):
        self._items = deque()

    def enqueue(self, item):
        """Добавить элемент в конец очереди - O(1)"""
        self._items.append(item)

    def dequeue(self):
        """Удалить элемент из начала очереди - O(1)"""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.popleft()

    def front(self):
        """Первый элемент очереди"""
        if self.is_empty():
            raise IndexError("front from empty queue")
        return self._items[0]

    def rear(self):
        """Последний элемент очереди"""
        if self.is_empty():
            raise IndexError("rear from empty queue")
        return self._items[-1]

    def is_empty(self):
        return len(self._items) == 0

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

Сравним производительность различных подходов:

```
import time
from collections import deque
def benchmark_queue_operations(n=100000):
    """Сравнение производительности операций с очередью"""

    # Тестирование списка (неэффективно)
    start_time = time.time()
    queue_list = []
    for i in range(n):
        queue_list.append(i)
    for i in range(n):
        queue_list.pop(0)
    list_time = time.time() - start_time
```

```
    # Тестирование deque (эффективно)
    start_time = time.time()
    queue_deque = deque()
    for i in range(n):
        queue_deque.append(i)
    for i in range(n):
        queue_deque.popleft()
    deque_time = time.time() - start_time

    print(f"Операций: {n}")
    print(f"Список: {list_time:.4f} секунд")
    print(f"Deque: {deque_time:.4f} секунд")
    print(f"Deque быстрее в {list_time/deque_time:.1f} раз")
# benchmark_queue_operations()
```

Практическое применение: симулятор обслуживания клиентов

Рассмотрим практический пример — симуляцию работы банковского отделения с несколькими окнами обслуживания:

```
import random
from collections import deque
class Customer:
    """Класс клиента банка"""

    def __init__(self, name, service_time):
        self.name = name
        self.service_time = service_time
        self.arrival_time = 0
        self.service_start = 0

    def __repr__(self):
        return f"Customer({self.name}, {self.service_time}min)"
class BankSimulator:
    """Симулятор работы банка"""

    def __init__(self, num_windows=2):
        self.queue = EfficientQueue()
        self.windows = [None] * num_windows  # None означает свободное окно
        self.window_timers = [0] * num_windows
        self.current_time = 0
        self.served_customers = []
        self.total_wait_time = 0

    def add_customer(self, customer):
        """Добавить клиента в очередь"""
        customer.arrival_time = self.current_time
        self.queue.enqueue(customer)
       print(f"Время {self.current_time}: {customer.name} встал в очередь")
```

```
    def process_minute(self):
        """Обработать одну минуту работы банка"""
        self.current_time += 1

        # Обновляем таймеры окон
        for i in range(len(self.windows)):
            if self.windows[i] is not None:
                self.window_timers[i] -= 1
                if self.window_timers[i] <= 0:
                    # Клиент обслужен
                    customer = self.windows[i]
                    self.served_customers.append(customer)
                     wait_time = customer.service_start - customer.arrival_time
                    self.total_wait_time += wait_time
                    print(f"Время {self.current_time}: {customer.name} "
                          f"обслужен (ждал {wait_time} мин)")
                    self.windows[i] = None

        # Назначаем клиентов на свободные окна
        for i in range(len(self.windows)):
            if self.windows[i] is None and not self.queue.is_empty():
                customer = self.queue.dequeue()
                customer.service_start = self.current_time
                self.windows[i] = customer
                self.window_timers[i] = customer.service_time
                print(f"Время {self.current_time}: {customer.name} "
                      f"начал обслуживание в окне {i+1}")

    def get_statistics(self):
        """Получить статистику работы"""
        if not self.served_customers:
            return "Клиенты не обслуживались"

        avg_wait = self.total_wait_time / len(self.served_customers)
        return (f"Обслужено клиентов: {len(self.served_customers)}\n"
                f"Среднее время ожидания: {avg_wait:.1f} минут\n"
                f"Клиентов в очереди: {self.queue.size()}")
# Симуляция работы банка
bank = BankSimulator(num_windows=2)
# Добавляем клиентов
customers_data = [
    ("Иван", 5), ("Мария", 3), ("Петр", 7),
    ("Анна", 2), ("Сергей", 4), ("Елена", 6)
]
for i, (name, service_time) in enumerate(customers_data):
    bank.add_customer(Customer(name, service_time))
    if i < len(customers_data) - 1:
        bank.process_minute()  # Клиенты приходят с интервалом в минуту
# Обрабатываем оставшееся время
for _ in range(20):  # Максимум 20 минут симуляции
```

```
    bank.process_minute()
    if bank.queue.is_empty() and all(w is None for w in bank.windows):
        break
print("\n" + "="*50)
print("ИТОГОВАЯ СТАТИСТИКА:")
print(bank.get_statistics())
```

Специализированные реализации в Python

Python предоставляет готовые реализации стеков и очередей в стандартной библиотеке:

```
from collections import deque
import queue
# deque как стек
stack_deque = deque()
stack_deque.append("first")    # push
stack_deque.append("second")   # push
print(stack_deque.pop())       # pop -> "second"
# deque как очередь
queue_deque = deque()
queue_deque.append("first")    # enqueue
queue_deque.append("second")   # enqueue
print(queue_deque.popleft())   # dequeue -> "first"
# Потокобезопасные очереди из модуля queue
thread_safe_queue = queue.Queue()        # FIFO очередь
lifo_queue = queue.LifoQueue()           # LIFO (стек)
priority_queue = queue.PriorityQueue()   # Очередь с приоритетами
# Пример приоритетной очереди
priority_queue.put((1, "высокий приоритет"))
priority_queue.put((3, "низкий приоритет"))
priority_queue.put((2, "средний приоритет"))
while not priority_queue.empty():
    priority, task = priority_queue.get()
    print(f"Приоритет {priority}: {task}")
```

Выбор конкретной реализации зависит от требований вашего приложения: для простых случаев достаточно списка или deque, для многопоточных приложений лучше использовать классы из модуля queue, а для специальных требований стоит создать собственную реализацию.

Основные принципы при работе со стеками и очередями

- Стеки идеальны для отслеживания состояния (например, история операций, рекурсивные вызовы).
- Очереди естественны для задач планирования и обработки в порядке поступления.
- Выбирайте эффективные структуры данных: deque вместо списка для очередей.
- Всегда проверяйте пустоту структуры перед извлечением элементов.

12.5. Деревья: иерархические данные

Деревья представляют собой одну из фундаментальных структур данных в информатике, позволяющую эффективно организовать иерархическую информацию. В отличие от линейных структур, таких как списки или массивы, деревья отражают естественные отношения подчинения и классификации, которые мы наблюдаем в реальном мире — от организационных структур компаний до файловых систем компьютера.

Дерево состоит из узлов (nodes), связанных отношениями «родитель-потомок». Каждый узел может иметь произвольное количество дочерних узлов, но только одного родительского узла. Узел без родителя называется корнем дерева (root), а узлы без потомков — листьями (leaves). Эта структура обеспечивает естественную иерархию и позволяет эффективно выполнять операции поиска, вставки и удаления.

Бинарные деревья: основа древовидных структур

Бинарное дерево — это специальный случай дерева, в котором каждый узел имеет не более двух потомков, традиционно называемых левым и правым дочерними узлами. Эта ограниченность не является недостатком, а наоборот — позволяет создать элегантные и эффективные алгоритмы для работы с данными.

Начнем с создания базовой структуры узла бинарного дерева:

```
class TreeNode:
    """Узел бинарного дерева"""

    def __init__(self, data):
        self.data = data          # Данные узла
        self.left = None          # Левый дочерний узел
        self.right = None         # Правый дочерний узел
        self.parent = None        # Родительский узел (опционально)

    def __str__(self):
        return str(self.data)

    def is_leaf(self):
        """Проверяет, является ли узел листом"""
        return self.left is None and self.right is None

    def has_left_child(self):
        """Проверяет наличие левого потомка"""
        return self.left is not None

    def has_right_child(self):
        """Проверяет наличие правого потомка"""
        return self.right is not None
```

Теперь создадим класс для работы с бинарным деревом:

```
class BinaryTree:
    """Класс для работы с бинарным деревом"""

    def __init__(self, root_data=None):
```

```
        if root_data is not None:
            self.root = TreeNode(root_data)
        else:
            self.root = None

    def is_empty(self):
        """Проверяет, пусто ли дерево"""
        return self.root is None

    def insert_left(self, parent_node, data):
        """Вставляет левый дочерний узел"""
        if parent_node.left is None:
            parent_node.left = TreeNode(data)
            parent_node.left.parent = parent_node
        else:
            # Если левый узел существует, делаем его потомком нового узла
            new_node = TreeNode(data)
            new_node.left = parent_node.left
            parent_node.left.parent = new_node
            parent_node.left = new_node
            new_node.parent = parent_node

    def insert_right(self, parent_node, data):
        """Вставляет правый дочерний узел"""
        if parent_node.right is None:
            parent_node.right = TreeNode(data)
            parent_node.right.parent = parent_node
        else:
            new_node = TreeNode(data)
            new_node.right = parent_node.right
            parent_node.right.parent = new_node
            parent_node.right = new_node
            new_node.parent = parent_node
```

Эта базовая реализация позволяет создавать и модифицировать структуру бинарного дерева. Однако истинная мощь деревьев раскрывается при использовании алгоритмов обхода, которые позволяют систематически посетить каждый узел дерева.

Алгоритмы обхода деревьев

Обход дерева (tree traversal) — это процесс посещения каждого узла дерева ровно один раз в определенном порядке. Существует несколько стандартных способов обхода бинарного дерева, каждый из которых имеет свое применение и особенности.

```
def preorder_traversal(self, node, visit_func=None):
    """
    Прямой (префиксный) обход: корень → левое поддерево → правое поддерево
    Применение: копирование дерева, получение префиксной записи выражения
    """
    if node is not None:
        if visit_func:
```

```
            visit_func(node.data)
        else:
            print(node.data, end=' ')

        self.preorder_traversal(node.left, visit_func)
        self.preorder_traversal(node.right, visit_func)
def inorder_traversal(self, node, visit_func=None):
    """
    Симметричный (инфиксный) обход: левое поддерево → корень → правое под-
дерево
    Применение: получение отсортированной последовательности в BST
    """
    if node is not None:
        self.inorder_traversal(node.left, visit_func)

        if visit_func:
            visit_func(node.data)
        else:
            print(node.data, end=' ')

        self.inorder_traversal(node.right, visit_func)
def postorder_traversal(self, node, visit_func=None):
    """
    Обратный (постфиксный) обход: левое поддерево → правое поддерево → корень
    Применение: удаление дерева, вычисление размера, постфиксная запись
    """
    if node is not None:
        self.postorder_traversal(node.left, visit_func)
        self.postorder_traversal(node.right, visit_func)

        if visit_func:
            visit_func(node.data)
        else:
            print(node.data, end=' ')
```

Добавим эти методы к нашему классу `BinaryTree` и продемонстрируем их работу:

```
# Расширяем класс BinaryTree
def get_height(self, node=None):
    """Вычисляет высоту дерева"""
    if node is None:
        node = self.root

    if node is None:
        return -1  # Высота пустого дерева

    if node.is_leaf():
        return 0

    left_height = self.get_height(node.left) if node.left else -1
```

```
    right_height = self.get_height(node.right) if node.right else -1

    return 1 + max(left_height, right_height)
def count_nodes(self, node=None):
    """Подсчитывает количество узлов в дереве"""
    if node is None:
        node = self.root

    if node is None:
        return 0

    return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)
# Демонстрация работы с бинарным деревом
tree = BinaryTree("A")
tree.insert_left(tree.root, "B")
tree.insert_right(tree.root, "C")
tree.insert_left(tree.root.left, "D")
tree.insert_right(tree.root.left, "E")
print("Прямой обход:")
tree.preorder_traversal(tree.root)  # A B D E C
print("\nСимметричный обход:")
tree.inorder_traversal(tree.root)   # D B E A C
print("\nОбратный обход:")
tree.postorder_traversal(tree.root) # D E B C A
```

Бинарные деревья поиска: эффективная организация данных

Бинарное дерево поиска (Binary Search Tree, BST) — это специальный тип бинарного дерева, в котором для каждого узла выполняется важное свойство: все значения в левом поддереве меньше значения текущего узла, а все значения в правом поддереве больше. Это свойство делает BST чрезвычайно эффективным для операций поиска, вставки и удаления.

```
class BinarySearchTree:
    """Бинарное дерево поиска"""

    def __init__(self):
        self.root = None
        self.size = 0

    def insert(self, data):
        """Вставляет элемент в дерево, сохраняя свойство BST"""
        if self.root is None:
            self.root = TreeNode(data)
        else:
            self._insert_recursive(self.root, data)
        self.size += 1

    def _insert_recursive(self, node, data):
        """Рекурсивная вставка элемента"""
        if data < node.data:
```

```
            if node.left is None:
                node.left = TreeNode(data)
                node.left.parent = node
            else:
                self._insert_recursive(node.left, data)
        elif data > node.data:
            if node.right is None:
                node.right = TreeNode(data)
                node.right.parent = node
            else:
                self._insert_recursive(node.right, data)
        # Если data == node.data, дубликаты не добавляем

    def search(self, data):
        """Поиск элемента в дереве"""
        return self._search_recursive(self.root, data)

    def _search_recursive(self, node, data):
        """Рекурсивный поиск элемента"""
        if node is None:
            return None

        if data == node.data:
            return node
        elif data < node.data:
            return self._search_recursive(node.left, data)
        else:
            return self._search_recursive(node.right, data)
```

Операция поиска в BST имеет среднюю временную сложность `O(log n)`, что значительно эффективнее линейного поиска `O(n)`. В худшем случае (вырожденное дерево, похожее на связанный список) сложность может достигать `O(n)`, но при сбалансированном дереве производительность остается логарифмической.

Реализуем операции для получения минимального и максимального элементов, а также удаления:

```
def find_min(self, node=None):
    """Находит минимальный элемент в дереве или поддереве"""
    if node is None:
        node = self.root

    if node is None:
        return None

    while node.left is not None:
        node = node.left
    return node
def find_max(self, node=None):
    """Находит максимальный элемент в дереве или поддереве"""
    if node is None:
        node = self.root
```

```
    if node is None:
        return None

    while node.right is not None:
        node = node.right
    return node
def delete(self, data):
    """Удаляет элемент из дерева"""
    node = self.search(data)
    if node:
        self._delete_node(node)
        self.size -= 1
def _delete_node(self, node):
    """Удаляет узел, сохраняя свойство BST"""
    # Случай 1: узел — лист
    if node.is_leaf():
        if node.parent:
            if node.parent.left == node:
                node.parent.left = None
            else:
                node.parent.right = None
        else:
            self.root = None

    # Случай 2: узел имеет одного потомка
    elif node.left is None:  # Только правый потомок
        self._replace_node(node, node.right)
    elif node.right is None:  # Только левый потомок
        self._replace_node(node, node.left)

    # Случай 3: узел имеет двух потомков
    else:
        # Находим преемника (минимальный элемент в правом поддереве)
        successor = self.find_min(node.right)
        node.data = successor.data
        self._delete_node(successor)
def _replace_node(self, old_node, new_node):
    """Заменяет один узел другим"""
    if old_node.parent:
        if old_node.parent.left == old_node:
            old_node.parent.left = new_node
        else:
            old_node.parent.right = new_node
    else:
        self.root = new_node

    if new_node:
        new_node.parent = old_node.parent
```

Практическое применение и анализ производительности

Бинарные деревья поиска находят широкое применение в различных алгоритмах и структурах данных. Они служат основой для реализации ассоциативных массивов, индексов баз данных, алгоритмов сжатия данных и многих других задач, требующих эффективного поиска и сортировки.

```
# Демонстрация работы BST
bst = BinarySearchTree()
elements = [50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45]
print("Вставляем элементы:", elements)
for elem in elements:
    bst.insert(elem)
print(f"Размер дерева: {bst.size}")
print(f"Высота дерева: {bst.get_height()}")
print("\nСимметричный обход (отсортированная последовательность):")
bst.inorder_traversal(bst.root)
print(f"\nМинимальный элемент: {bst.find_min().data}")
print(f"Максимальный элемент: {bst.find_max().data}")
# Поиск элементов
search_values = [45, 75, 10]
for value in search_values:
    result = bst.search(value)
    if result:
        print(f"Элемент {value} найден")
    else:
        print(f"Элемент {value} не найден")
# Удаление элементов
print("\nУдаляем элемент 30")
bst.delete(30)
print("Дерево после удаления:")
bst.inorder_traversal(bst.root)
```

Временная сложность основных операций в BST составляет:

- **Поиск:** `O(log n)` в среднем, `O(n)` в худшем случае
- **Вставка:** `O(log n)` в среднем, `O(n)` в худшем случае
- **Удаление:** `O(log n)` в среднем, `O(n)` в худшем случае
- **Обход:** `O(n)` для любого типа обхода

Пространственная сложность равна `O(n)` для хранения n элементов плюс `O(log n)` для стека рекурсивных вызовов в сбалансированном дереве.

Понимание деревьев и их алгоритмов является фундаментальным для многих продвинутых структур данных и алгоритмов. Концепции, изученные в этом разделе, служат основой для понимания более сложных древовидных структур, таких как AVL-деревья, красно-черные деревья и B-деревья, которые используются в промышленных системах управления базами данных и файловых системах.

Глава 13. Специализированные области

13.1. Веб-разработка: Flask как введение

Веб-разработка представляет собой одну из наиболее востребованных и динамично развивающихся областей программирования. В отличие от создания консольных приложений, которые мы изучали ранее, веб-приложения работают по принципу «клиент-сервер», где пользователи взаимодействуют с программой через браузер, отправляя HTTP-запросы и получая HTML-страницы в ответ.

Flask — это микрофреймворк для Python, который позволяет быстро создавать веб-приложения без излишней сложности. В отличие от более крупных фреймворков, Flask придерживается философии минимализма, предоставляя только необходимые инструменты и позволяя разработчику самостоятельно выбирать дополнительные компоненты. Эта особенность делает Flask идеальным инструментом для изучения основ веб-разработки, поскольку вы понимаете каждый компонент системы.

Установка Flask и первое приложение

Для работы с Flask необходимо установить его через менеджер пакетов `pip`. Рекомендуется создать виртуальное окружение для изоляции зависимостей проекта, как мы изучали в главе о модулях и пакетах:

```
# В командной строке
python -m venv flask_env
# Активация окружения (Windows)
flask_env\Scripts\activate
# Активация окружения (Linux/Mac)
source flask_env/bin/activate
# Установка Flask
pip install Flask
```

Создадим наше первое веб-приложение, которое демонстрирует основные принципы работы Flask:

```
# app.py
from flask import Flask
# Создание экземпляра приложения
app = Flask(__name__)
# Декоратор route определяет URL-адрес, по которому будет доступна функция
@app.route('/')
def home():
    """Главная страница приложения"""
    return '<h1>Добро пожаловать в мир Flask!</h1>'
@app.route('/about')
def about():
    """Страница с информацией о приложении"""
    return '''
```

```
    <h1>О приложении</h1>
    <p>Это простое веб-приложение на Flask для изучения основ веб-разработки.</p>
    <a href="/">Вернуться на главную</a>
    '''
# Запуск приложения в режиме отладки
if __name__ == '__main__':
    app.run(debug=True)
```

При запуске этого скрипта Flask создаст локальный веб-сервер, доступный по адресу `http://127.0.0.1:5000`. Параметр `debug=True` включает режим отладки, который автоматически перезапускает сервер при изменении кода и предоставляет подробную информацию об ошибках.

Маршруты: навигация в веб-приложении

Маршруты (routes) в Flask определяют соответствие между URL-адресами и функциями Python. Это основной механизм, который позволяет создавать структурированные веб-приложения с множеством страниц и функций:

```
# routes_example.py
from flask import Flask
app = Flask(__name__)
# Статический маршрут
@app.route('/')
def index():
    return '<h1>Главная страница</h1>'
# Маршрут с параметром
@app.route('/user/<username>')
def user_profile(username):
    return f'<h1>Профиль пользователя: {username}</h1>'
# Маршрут с типизированным параметром
@app.route('/post/<int:post_id>')
def show_post(post_id):
    # Здесь post_id автоматически преобразуется в целое число
    return f'<h1>Статья номер {post_id}</h1>'
# Маршрут, поддерживающий несколько HTTP-методов
@app.route('/data', methods=['GET', 'POST'])
def handle_data():
    from flask import request
    if request.method == 'POST':
        return '<h1>Данные получены методом POST</h1>'
    return '<h1>Страница для просмотра данных</h1>'
# Маршрут с необязательным параметром
@app.route('/articles/')
@app.route('/articles/<category>')
def articles(category=None):
    if category:
        return f'<h1>Статьи в категории: {category}</h1>'
    return '<h1>Все статьи</h1>'
if __name__ == '__main__':
    app.run(debug=True)
```

Flask поддерживает несколько типов параметров в маршрутах: `<string>` (по умолчанию), `<int>` для целых чисел, `<float>` для чисел с плавающей точкой, `<path>` для путей с символами /, и `<uuid>` для UUID. Эта типизация обеспечивает автоматическую валидацию и преобразование параметров.

Шаблоны: разделение логики и представления

Возвращение HTML-кода непосредственно из функций Python нарушает принцип разделения ответственности и делает код трудно поддерживаемым. Flask использует систему шаблонов Jinja2, которая позволяет создавать динамический HTML с использованием специального синтаксиса.

Создадим структуру каталогов для работы с шаблонами:

```
project/
├── app.py
├── templates/
│   ├── base.html
│   ├── index.html
│   └── user.html
└── static/
    ├── style.css
    └── script.js
```

Базовый шаблон определяет общую структуру страниц:

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Мое Flask-приложение{% endblock %}</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css')
}}">
</head>
<body>
    <nav>
        <ul>
            <li><a href="{{ url_for('index') }}">Главная</a></li>
            <li><a href="{{ url_for('about') }}">О нас</a></li>
        </ul>
    </nav>

    <main>
        {% block content %}{% endblock %}
    </main>

    <footer>
        <p>&copy; 2024 Мое Flask-приложение</p>
    </footer>
</body>
</html>
```

Дочерние шаблоны наследуют структуру от базового и определяют специфичное содержимое:

```
<!-- templates/index.html -->
{% extends "base.html" %}
{% block title %}Главная страница{% endblock %}
{% block content %}
    <h1>Добро пожаловать!</h1>
    <p>Текущее время: {{ current_time }}</p>

    {% if users %}
        <h2>Пользователи онлайн:</h2>
        <ul>
        {% for user in users %}
            <li>
               <a href="{{ url_for('user_profile', username=user.name) }}">
                    {{ user.name }} ({{ user.role }})
                </a>
            </li>
        {% endfor %}
        </ul>
    {% else %}
        <p>Нет пользователей онлайн</p>
    {% endif %}
{% endblock %}
```

Python-код для работы с шаблонами:

```
# templates_app.py
from flask import Flask, render_template
from datetime import datetime
app = Flask(__name__)
# Имитация базы данных пользователей
users_data = [
    {'name': 'alice', 'role': 'admin'},
    {'name': 'bob', 'role': 'user'},
    {'name': 'charlie', 'role': 'moderator'}
]
class User:
    def __init__(self, name, role):
        self.name = name
        self.role = role
@app.route('/')
def index():
    # Создаем объекты пользователей
    users = [User(user['name'], user['role']) for user in users_data]

    return render_template('index.html',
                           current_time=datetime.now().strftime('%H:%M:%S'),
                           users=users)
@app.route('/user/<username>')
def user_profile(username):
```

```
    # Поиск пользователя в "базе данных"
      user_info = next((user for user in users_data if user['name'] ==
username), None)

    if not user_info:
        return render_template('404.html'), 404

    return render_template('user.html', user=user_info)
@app.route('/about')
def about():
    return render_template('about.html')
if __name__ == '__main__':
    app.run(debug=True)
```

Работа с формами

Формы являются основным способом получения данных от пользователей в веб-приложениях. Flask предоставляет удобные инструменты для обработки данных форм и валидации пользовательского ввода:

```
<!-- templates/contact.html -->
{% extends "base.html" %}
{% block title %}Связаться с нами{% endblock %}
{% block content %}
    <h1>Форма обратной связи</h1>

    {% if message %}
        <div class="alert alert-success">{{ message }}</div>
    {% endif %}

    {% if errors %}
        <div class="alert alert-error">
            <ul>
            {% for error in errors %}
                <li>{{ error }}</li>
            {% endfor %}
            </ul>
        </div>
    {% endif %}

    <form method="POST">
        <div class="form-group">
            <label for="name">Имя:</label>
           <input type="text" id="name" name="name" value="{{ request.form.
name }}" required>
        </div>

        <div class="form-group">
            <label for="email">Email:</label>
             <input type="email" id="email" name="email" value="{{ request.
form.email }}" required>
        </div>
```

```
        <div class="form-group">
            <label for="subject">Тема:</label>
            <select id="subject" name="subject">
                <option value="general">Общий вопрос</option>
                <option value="support">Техническая поддержка</option>
                <option value="feedback">Отзыв</option>
            </select>
        </div>

        <div class="form-group">
            <label for="message">Сообщение:</label>
            <textarea id="message" name="message" rows="5">{{ request.form.
message }}</textarea>
        </div>

        <button type="submit">Отправить</button>
    </form>
{% endblock %}
```

Обработка форм на стороне сервера:

```
# forms_app.py
from flask import Flask, render_template, request, redirect, url_for, flash
import re
app = Flask(__name__)
app.secret_key = 'your-secret-key-here'  # Необходим для работы с flash-со-
общениями
def validate_email(email):
    """Простая валидация email-адреса"""
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email) is not None
def validate_contact_form(form_data):
    """Валидация данных формы обратной связи"""
    errors = []

    if not form_data.get('name', '').strip():
        errors.append('Имя обязательно для заполнения')
    elif len(form_data['name']) < 2:
        errors.append('Имя должно содержать минимум два символа')

    email = form_data.get('email', '').strip()
    if not email:
        errors.append('Email обязателен для заполнения')
    elif not validate_email(email):
        errors.append('Некорректный формат email-адреса')

    if not form_data.get('message', '').strip():
        errors.append('Сообщение не может быть пустым')
    elif len(form_data['message']) < 10:
        errors.append('Сообщение должно содержать минимум 10 символов')
```

```
    return errors
@app.route('/contact', methods=['GET', 'POST'])
def contact():
    if request.method == 'POST':
        # Валидация данных формы
        errors = validate_contact_form(request.form)

        if errors:
            return render_template('contact.html', errors=errors)

        # Обработка успешной отправки формы
         # В реальном приложении здесь были бы отправка email или сохране-
ние в базу данных
        contact_data = {
            'name': request.form['name'],
            'email': request.form['email'],
            'subject': request.form['subject'],
            'message': request.form['message']
        }

        # Имитация сохранения данных
        save_contact_message(contact_data)

          flash('Спасибо за ваше сообщение! Мы свяжемся с вами в ближайшее
время.', 'success')
        return redirect(url_for('contact'))

    return render_template('contact.html')
def save_contact_message(data):
    """Имитация сохранения сообщения (в реальности - в базу данных)"""
    import json
    import os
    from datetime import datetime

    # Добавляем временную метку
    data['timestamp'] = datetime.now().isoformat()

    # Сохраняем в JSON-файл (в реальном проекте использовалась бы база дан-
ных)
    if not os.path.exists('messages.json'):
        messages = []
    else:
        with open('messages.json', 'r', encoding='utf-8') as f:
            messages = json.load(f)

    messages.append(data)

    with open('messages.json', 'w', encoding='utf-8') as f:
        json.dump(messages, f, ensure_ascii=False, indent=2)
if __name__ == '__main__':
    app.run(debug=True)
```

REST API: создание веб-сервисов

REST API (Representational State Transfer Application Programming Interface) позволяет создавать веб-сервисы, которые могут взаимодействовать с другими приложениями, мобильными клиентами или фронтенд-фреймворками через HTTP-запросы, возвращая данные в формате JSON:

```
# api_app.py
from flask import Flask, jsonify, request
from datetime import datetime
import uuid
app = Flask(__name__)
# Имитация базы данных в памяти
books_db = {
    '1': {'id': '1', 'title': 'Python для начинающих', 'author': 'Иван Ива-
нов', 'year': 2023},
    '2': {'id': '2', 'title': 'Веб-разработка с Flask', 'author': 'Мария Пе-
трова', 'year': 2024}
}
def create_error_response(message, status_code):
    """Создание стандартного ответа об ошибке"""
    return jsonify({'error': message}), status_code
def validate_book_data(data):
    """Валидация данных книги"""
    required_fields = ['title', 'author', 'year']
    errors = []

    for field in required_fields:
        if field not in data or not data[field]:
            errors.append(f'Поле "{field}" обязательно для заполнения')

    if 'year' in data:
        try:
            year = int(data['year'])
            current_year = datetime.now().year
            if year < 1000 or year > current_year + 1:
             errors.append('Год должен быть между 1000 и текущим годом + 1')
        except (ValueError, TypeError):
            errors.append('Год должен быть числом')

    return errors
# GET /api/books - получение списка всех книг
@app.route('/api/books', methods=['GET'])
def get_books():
    """Получение списка всех книг"""
    books_list = list(books_db.values())
    return jsonify({
        'books': books_list,
        'total': len(books_list)
    })
# GET /api/books/<id> - получение книги по ID
@app.route('/api/books/<book_id>', methods=['GET'])
```

```
def get_book(book_id):
    """Получение книги по ID"""
    book = books_db.get(book_id)
    if not book:
        return create_error_response('Книга не найдена', 404)

    return jsonify(book)
# POST /api/books - создание новой книги
@app.route('/api/books', methods=['POST'])
def create_book():
    """Создание новой книги"""
    if not request.json:
         return create_error_response('Данные должны быть в формате JSON',
400)

    errors = validate_book_data(request.json)
    if errors:
        return jsonify({'errors': errors}), 400

    # Создание новой книги
    book_id = str(uuid.uuid4())
    book = {
        'id': book_id,
        'title': request.json['title'],
        'author': request.json['author'],
        'year': int(request.json['year']),
        'created_at': datetime.now().isoformat()
    }

    books_db[book_id] = book
    return jsonify(book), 201
# PUT /api/books/<id> - обновление книги
@app.route('/api/books/<book_id>', methods=['PUT'])
def update_book(book_id):
    """Обновление существующей книги"""
    if book_id not in books_db:
        return create_error_response('Книга не найдена', 404)

    if not request.json:
         return create_error_response('Данные должны быть в формате JSON',
400)

    errors = validate_book_data(request.json)
    if errors:
        return jsonify({'errors': errors}), 400

    # Обновление книги
    book = books_db[book_id]
    book.update({
        'title': request.json['title'],
        'author': request.json['author'],
```

```
        'year': int(request.json['year']),
        'updated_at': datetime.now().isoformat()
    })

    return jsonify(book)
# DELETE /api/books/<id> - удаление книги
@app.route('/api/books/<book_id>', methods=['DELETE'])
def delete_book(book_id):
    """Удаление книги по ID"""
    if book_id not in books_db:
        return create_error_response('Книга не найдена', 404)

    deleted_book = books_db.pop(book_id)
    return jsonify({
        'message': 'Книга успешно удалена',
        'deleted_book': deleted_book
    })
# Обработчик ошибок 404
@app.errorhandler(404)
def not_found(error):
    return create_error_response('Ресурс не найден', 404)
# Обработчик ошибок 500
@app.errorhandler(500)
def internal_error(error):
    return create_error_response('Внутренняя ошибка сервера', 500)
if __name__ == '__main__':
    app.run(debug=True)
```

Для тестирования API можно использовать команды `curl` или создать простой клиент:

```
# api_client.py
import requests
import json
BASE_URL = 'http://127.0.0.1:5000/api'
def test_api():
    """Простое тестирование API"""

    # Получение списка книг
    print("1. Получение списка книг:")
    response = requests.get(f'{BASE_URL}/books')
    print(f"Статус: {response.status_code}")
    print(f"Ответ: {response.json()}\n")

    # Создание новой книги
    print("2. Создание новой книги:")
    new_book = {
        'title': 'Алгоритмы и структуры данных',
        'author': 'Александр Сидоров',
        'year': 2024
    }
```

```
    response = requests.post(f'{BASE_URL}/books', json=new_book)
    print(f"Статус: {response.status_code}")
    created_book = response.json()
    print(f"Ответ: {created_book}\n")

    # Получение созданной книги
    if response.status_code == 201:
        book_id = created_book['id']
        print(f"3. Получение книги с ID {book_id}:")
        response = requests.get(f'{BASE_URL}/books/{book_id}')
        print(f"Статус: {response.status_code}")
        print(f"Ответ: {response.json()}\n")

    # Попытка получить несуществующую книгу
    print("4. Попытка получить несуществующую книгу:")
    response = requests.get(f'{BASE_URL}/books/nonexistent')
    print(f"Статус: {response.status_code}")
    print(f"Ответ: {response.json()}")
if __name__ == '__main__':
    test_api()
```

Flask предоставляет мощные инструменты для создания как традиционных веб-приложений с HTML-интерфейсом, так и современных API-сервисов. Понимание этих основ открывает путь к созданию полноценных веб-приложений, которые могут служить фундаментом для более сложных проектов. В следующих разделах мы изучим другие специализированные области Python, которые дополнят ваши навыки веб-разработки знаниями в области анализа данных и машинного обучения.

13.2. Анализ данных: pandas и numpy

Современный мир перенасыщен данными, и умение эффективно их обрабатывать и анализировать стало одним из наиболее востребованных навыков. Python предоставляет мощную экосистему для работы с данными, в основе которой лежат две фундаментальные библиотеки: **numpy** для численных вычислений и **pandas** для работы со структурированными данными.

NumPy (Numerical Python) — это основа для научных вычислений в Python. Библиотека предоставляет эффективные многомерные массивы и математические функции для работы с ними. **Pandas** построен поверх NumPy и предлагает высокоуровневые структуры данных и инструменты анализа, делая работу с табличными данными интуитивно понятной.

Установка и первое знакомство

Прежде чем начать работу, необходимо установить библиотеки. В большинстве случаев они уже входят в дистрибутивы типа Anaconda, но при необходимости можно установить их через pip:

```
pip install pandas numpy
```

Начнем с импорта библиотек и изучения основных структур данных:

```
import pandas as pd
import numpy as np
# Создание простого массива numpy
arr = np.array([1, 2, 3, 4, 5])
print("NumPy array:", arr)
print("Тип данных:", arr.dtype)
# Создание DataFrame из словаря
data = {
    'name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'age': [25, 30, 35, 28],
    'salary': [50000, 60000, 70000, 55000]
}
df = pd.DataFrame(data)
print("\nDataFrame:")
print(df)
```

DataFrame — это центральная структура данных pandas, представляющая собой двумерную таблицу с индексированными строками и столбцами. Это аналог электронной таблицы или таблицы базы данных, но с расширенными возможностями для анализа и обработки данных.

Загрузка данных из файлов

В реальной работе данные редко создаются вручную — чаще всего их загружают из внешних источников. Pandas предоставляет удобные функции для чтения различных форматов:

```
# Создадим тестовый CSV-файл
test_data = """name,age,department,salary
John,28,IT,65000
Sarah,32,Marketing,58000
Mike,24,IT,52000
Emma,29,Sales,61000
David,35,Marketing,72000"""
with open('employees.csv', 'w') as f:
    f.write(test_data)
# Загрузка данных из CSV
df = pd.read_csv('employees.csv')
print("Данные из CSV-файла:")
print(df)
print(f"\nРазмер данных: {df.shape}")  # (строки, столбцы)
```

Функция `read_csv()` — это универсальный инструмент для работы с табличными данными. Она автоматически определяет типы данных, обрабатывает заголовки и предоставляет множество параметров для настройки процесса загрузки.

Исследование структуры данных

Прежде чем приступить к анализу, важно понимать структуру и качество данных. Pandas предоставляет несколько методов для быстрой оценки датасета:

```
# Основная информация о данных
print("Первые несколько строк:")
print(df.head())
print("\nПоследние несколько строк:")
print(df.tail(3))
print("\nИнформация о структуре:")
print(df.info())
print("\nОписательная статистика:")
print(df.describe())
```

Метод `info()` предоставляет общую информацию о `DataFrame`: количество строк, названия столбцов, типы данных и количество непустых значений. Метод `describe()` генерирует описательную статистику для численных столбцов, включая среднее значение, стандартное отклонение, минимум, максимум и квартили.

Индексация и выборка данных

Работа с данными часто начинается с выборки нужных фрагментов. `Pandas` предлагает несколько способов индексации и фильтрации данных:

```
# Выбор одного столбца
names = df['name']
print("Имена сотрудников:")
print(names)
# Выбор нескольких столбцов
subset = df[['name', 'salary']]
print("\nИмена и зарплаты:")
print(subset)
# Фильтрация данных по условию
high_earners = df[df['salary'] > 60000]
print("\nСотрудники с зарплатой выше 60000:")
print(high_earners)
# Сложные условия
it_high_earners = df[(df['department'] == 'IT') & (df['salary'] > 50000)]
print("\nIT-сотрудники с высокой зарплатой:")
print(it_high_earners)
```

`Pandas` использует логическую индексацию, позволяющую создавать сложные фильтры с помощью логических операторов. Важно помнить, что для комбинирования условий нужно использовать операторы `&` (и) и `|` (или) вместо `and` и `or`.

Группировка и агрегация данных

Одна из самых мощных возможностей `pandas` — группировка данных для вычисления агрегированных показателей. Это позволяет отвечать на вопросы типа «Какая средняя зарплата в каждом отделе?»:

```
# Группировка по отделам
dept_stats = df.groupby('department').agg({
    'salary': ['mean', 'min', 'max', 'count'],
    'age': 'mean'
})
```

```
print("Статистика по отделам:")
print(dept_stats)
# Более простая группировка
avg_salary_by_dept = df.groupby('department')['salary'].mean()
print("\nСредняя зарплата по отделам:")
print(avg_salary_by_dept)
```

Метод `groupby()` создает группированный объект, к которому можно применять различные агрегирующие функции. Метод `agg()` позволяет применять разные функции к разным столбцам одновременно, что очень удобно для комплексного анализа.

Обработка пропущенных значений

В реальных данных часто встречаются пропущенные значения, и правильная их обработка критически важна для качества анализа:

```
# Создадим данные с пропущенными значениями
data_with_na = {
    'product': ['A', 'B', 'C', 'D', 'E'],
    'price': [100, None, 150, 200, None],
    'quantity': [10, 20, None, 15, 25]
}
df_na = pd.DataFrame(data_with_na)
print("Данные с пропусками:")
print(df_na)
# Проверка наличия пропущенных значений
print("\nИнформация о пропусках:")
print(df_na.isnull().sum())
# Заполнение пропущенных значений
df_filled = df_na.fillna({
    'price': df_na['price'].mean(),
    'quantity': df_na['quantity'].median()
})
print("\nДанные после заполнения пропусков:")
print(df_filled)
```

Pandas предоставляет несколько стратегий для работы с пропущенными значениями: удаление строк или столбцов с пропусками (`dropna()`), заполнение значениями (`fillna()`) или интерполяция. Выбор стратегии зависит от природы данных и задач анализа.

Создание новых столбцов и трансформации

Часто требуется создавать новые переменные на основе существующих или трансформировать данные для анализа:

```
# Загрузим исходные данные о сотрудниках
df = pd.read_csv('employees.csv')
# Создание нового столбца
df['salary_category'] = df['salary'].apply(
    lambda x: 'High' if x > 60000 else 'Medium' if x > 55000 else 'Low'
)
```

```
# Создание столбца на основе условий
df['experience_level'] = np.where(df['age'] > 30, 'Senior', 'Junior')
# Вычисление производных показателей
df['monthly_salary'] = df['salary'] / 12
print("Данные с новыми столбцами:")
print(df)
```

Функция `apply()` позволяет применять пользовательские функции к данным, а `np.where()` реализует векторизованные условные операции. Эти инструменты дают гибкость в создании новых признаков для анализа.

Базовые статистические операции

Pandas интегрирует множество статистических функций, позволяя быстро получать важные показатели:

```
# Корреляционный анализ численных переменных
numeric_df = df.select_dtypes(include=[np.number])
correlation_matrix = numeric_df.corr()
print("Корреляционная матрица:")
print(correlation_matrix)
# Базовые статистики
print("\nОсновные статистические показатели:")
print(f"Средняя зарплата: {df['salary'].mean():.2f}")
print(f"Медианная зарплата: {df['salary'].median():.2f}")
print(f"Стандартное отклонение: {df['salary'].std():.2f}")
print(f"Размах зарплат: {df['salary'].max() - df['salary'].min()}")
# Частотный анализ категориальных переменных
dept_counts = df['department'].value_counts()
print("\nРаспределение по отделам:")
print(dept_counts)
```

Эти функции позволяют быстро оценить основные характеристики данных, выявить паттерны и аномалии, что критически важно на начальных этапах анализа.

Простая визуализация данных

Хотя детальное изучение визуализации выходит за рамки этой главы, pandas предоставляет базовые возможности построения графиков:

```
import matplotlib.pyplot as plt
# Гистограмма распределения зарплат
df['salary'].hist(bins=10, alpha=0.7)
plt.title('Distribution of Salaries')
plt.xlabel('Salary')
plt.ylabel('Frequency')
plt.show()
# Столбчатая диаграмма по отделам
dept_counts.plot(kind='bar')
plt.title('Employees by Department')
plt.xlabel('Department')
plt.ylabel('Count')
```

```
plt.xticks(rotation=45)
plt.show()
```

Визуализация помогает лучше понимать данные, выявлять тенденции и представлять результаты анализа в наглядной форме.

Практический пример: анализ продаж

Объединим изученные концепции в комплексном примере анализа данных о продажах:

```
# Создание расширенного датасета продаж
sales_data = {
    'date': pd.date_range('2023-01-01', periods=100, freq='D'),
    'product': np.random.choice(['Laptop', 'Phone', 'Tablet'], 100),
    'region': np.random.choice(['North', 'South', 'East', 'West'], 100),
    'sales_amount': np.random.normal(1000, 300, 100),
    'quantity': np.random.randint(1, 10, 100)
}
sales_df = pd.DataFrame(sales_data)
sales_df['sales_amount'] = np.maximum(sales_df['sales_amount'], 100)  # Ми-
нимум 100
print("Первые строки данных о продажах:")
print(sales_df.head())
# Добавление производных переменных
sales_df['unit_price'] = sales_df['sales_amount'] / sales_df['quantity']
sales_df['month'] = sales_df['date'].dt.month
sales_df['weekday'] = sales_df['date'].dt.day_name()
# Комплексный анализ
print("\nАнализ продаж по продуктам:")
product_analysis = sales_df.groupby('product').agg({
    'sales_amount': ['sum', 'mean', 'count'],
    'quantity': 'sum',
    'unit_price': 'mean'
}).round(2)
print(product_analysis)
```

Этот пример демонстрирует типичный workflow анализа данных: загрузка, очистка, создание новых переменных и агрегированный анализ.

Временные ряды и даты

Pandas обладает мощными возможностями для работы с временными данными:

```
# Анализ продаж по времени
monthly_sales = sales_df.groupby('month')['sales_amount'].sum()
print("Продажи по месяцам:")
print(monthly_sales)
# Анализ по дням недели
weekday_sales = sales_df.groupby('weekday')['sales_amount'].mean()
print("\nСредние продажи по дням недели:")
print(weekday_sales)
```

```
# Нахождение топ-продаж
top_sales_days = sales_df.nlargest(5, 'sales_amount')[['date', 'product',
'sales_amount']]
print("\nТоп-5 дней по продажам:")
print(top_sales_days)
```

Работа с датами и временными рядами — важная составляющая многих аналитических задач, от финансового анализа до изучения пользовательского поведения.

Резюме и лучшие практики

Pandas и NumPy предоставляют мощный инструментарий для анализа данных. Основные принципы эффективной работы таковы: всегда начинать с исследования структуры данных (`info()`, `describe()`, `head()`), проверять качество данных и обрабатывать пропущенные значения, использовать векторизованные операции вместо циклов для повышения производительности и создавать осмысленные визуализации для лучшего понимания данных.

При работе с большими датасетами важно оптимизировать использование памяти, выбирая подходящие типы данных и загружая только необходимые столбцы. Pandas предоставляет параметры для контроля потребления памяти и производительности операций.

Освоение этих основ открывает путь к более сложным видам анализа: машинному обучению, временным рядам, статистическому моделированию и созданию интерактивных визуализаций. В следующем разделе мы рассмотрим, как применить полученные навыки для построения моделей машинного обучения с помощью библиотеки `scikit-learn`.

13.3. Машинное обучение: scikit-learn

Машинное обучение (machine learning) представляет собой область искусственного интеллекта, где алгоритмы автоматически находят закономерности в данных и делают прогнозы. Python стал доминирующим языком в этой сфере благодаря мощным библиотекам, среди которых `scikit-learn` занимает особое место как наиболее доступный инструмент для начинающих.

Библиотека `scikit-learn` предоставляет единообразный интерфейс для сотен алгоритмов машинного обучения. Ее философия основана на принципах простоты использования, производительности и совместимости с другими библиотеками научных вычислений Python.

Подготовка рабочей среды

Для работы с машинным обучением потребуется установить несколько библиотек:

```
# Установка через pip (выполнить в терминале)
# pip install scikit-learn pandas matplotlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, mean_squared_error
```

Основная концепция машинного обучения построена на разделении данных на признаки (features) — входные переменные, и целевую переменную (target) — то, что мы хотим предсказать. В `scikit-learn` это традиционно обозначается как `X` и `y` соответственно.

Классификация: определение категорий

Классификация решает задачу отнесения объекта к одному из заранее известных классов. Рассмотрим классический пример — определение вида ириса по параметрам цветка:

```
from sklearn import datasets
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
# Загружаем встроенный dataset ирисов
iris = datasets.load_iris()
X, y = iris.data, iris.target
# Разделяем данные на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
print(f"Размер обучающей выборки: {X_train.shape}")
print(f"Размер тестовой выборки: {X_test.shape}")
print(f"Классы: {iris.target_names}")
```

Базовый алгоритм классификации — логистическая регрессия (logistic regression). Несмотря на название, это метод классификации, использующий логистическую функцию для определения вероятностей принадлежности к классам:

```
# Создаем модель логистической регрессии
log_reg = LogisticRegression(random_state=42, max_iter=200)
# Обучаем модель на тренировочных данных
log_reg.fit(X_train, y_train)
# Делаем предсказания на тестовых данных
y_pred_lr = log_reg.predict(X_test)
# Вычисляем точность модели
accuracy_lr = accuracy_score(y_test, y_pred_lr)
print(f"Точность логистической регрессии: {accuracy_lr:.3f}")
# Посмотрим на несколько предсказаний
for i in range(5):
    actual = iris.target_names[y_test[i]]
    predicted = iris.target_names[y_pred_lr[i]]
    print(f"Фактический класс: {actual}, Предсказанный: {predicted}")
```

Случайный лес (Random Forest) — более сложный алгоритм, который строит множество деревьев решений и усредняет их результаты. Это один из наиболее надежных алгоритмов для начинающих:

```
# Создаем модель случайного леса
rf_classifier = RandomForestClassifier(
    n_estimators=100,  # количество деревьев в лесу
    random_state=42
)
```

```
# Обучаем модель
rf_classifier.fit(X_train, y_train)
# Делаем предсказания
y_pred_rf = rf_classifier.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Точность случайного леса: {accuracy_rf:.3f}")
# Важность признаков
feature_importance = rf_classifier.feature_importances_
for i, importance in enumerate(feature_importance):
    print(f"{iris.feature_names[i]}: {importance:.3f}")
```

Регрессия: предсказание непрерывных значений

Регрессия решает задачу предсказания числового значения. Создадим пример с предсказанием цены дома на основе его характеристик:

```
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
# Создаем синтетический dataset для регрессии
X_reg, y_reg = make_regression(
    n_samples=1000,     # количество примеров
    n_features=5,       # количество признаков
    n_informative=3,    # количество информативных признаков
    noise=10,           # уровень шума
    random_state=42
)
# Разделяем данные
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(
    X_reg, y_reg, test_size=0.2, random_state=42
)
print(f"Диапазон целевых значений: {y_reg.min():.2f} - {y_reg.max():.2f}")
```

Линейная регрессия (linear regression) — основной алгоритм для задач регрессии. Она предполагает линейную зависимость между признаками и целевой переменной:

```
# Масштабирование признаков для лучшей работы алгоритмов
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_reg)
X_test_scaled = scaler.transform(X_test_reg)
# Создаем модель линейной регрессии
linear_reg = LinearRegression()
linear_reg.fit(X_train_scaled, y_train_reg)
# Делаем предсказания
y_pred_linear = linear_reg.predict(X_test_scaled)
# Вычисляем среднеквадратичную ошибку
mse_linear = mean_squared_error(y_test_reg, y_pred_linear)
rmse_linear = np.sqrt(mse_linear)
print(f"RMSE линейной регрессии: {rmse_linear:.2f}")
# Коэффициенты модели показывают важность признаков
print("Коэффициенты модели:")
```

```
for i, coef in enumerate(linear_reg.coef_):
    print(f"Признак {i}: {coef:.3f}")
```

Случайный лес также эффективен для задач регрессии, особенно когда зависимость между переменными нелинейная:

```
# Модель случайного леса для регрессии
rf_regressor = RandomForestRegressor(
    n_estimators=100,
    random_state=42
)
rf_regressor.fit(X_train_reg, y_train_reg)
y_pred_rf_reg = rf_regressor.predict(X_test_reg)
mse_rf = mean_squared_error(y_test_reg, y_pred_rf_reg)
rmse_rf = np.sqrt(mse_rf)
print(f"RMSE случайного леса: {rmse_rf:.2f}")
# Сравниваем результаты
print(f"\nСравнение моделей:")
print(f"Линейная регрессия: {rmse_linear:.2f}")
print(f"Случайный лес: {rmse_rf:.2f}")
```

Кластеризация: поиск скрытых групп

Кластеризация — это обучение без учителя (unsupervised learning), где алгоритм самостоятельно находит группы похожих объектов в данных без заранее известных ответов:

```
from sklearn.cluster import KMeans, DBSCAN
from sklearn.datasets import make_blobs
# Создаем данные с естественными кластерами
X_clusters, _ = make_blobs(
    n_samples=300,      # количество точек
    centers=4,          # количество центров кластеров
    n_features=2,       # двумерные данные для визуализации
    random_state=42,
    cluster_std=1.0     # стандартное отклонение кластеров
)
print(f"Форма данных: {X_clusters.shape}")
```

K-means — наиболее популярный алгоритм кластеризации. Он разделяет данные на заранее заданное количество кластеров:

```
# Создаем модель K-means
kmeans = KMeans(n_clusters=4, random_state=42, n_init=10)
# Выполняем кластеризацию
cluster_labels = kmeans.fit_predict(X_clusters)
# Получаем центры кластеров
cluster_centers = kmeans.cluster_centers_
print("Центры кластеров:")
for i, center in enumerate(cluster_centers):
    print(f"Кластер {i}: ({center[0]:.2f}, {center[1]:.2f})")
# Подсчитываем размеры кластеров
```

```
unique, counts = np.unique(cluster_labels, return_counts=True)
for cluster, count in zip(unique, counts):
    print(f"Кластер {cluster}: {count} точек")
```

DBSCAN (Density-Based Spatial Clustering) — алгоритм, основанный на плотности. В отличие от K-means, он может находить кластеры произвольной формы и автоматически определять их количество:

```
# Создаем модель DBSCAN
dbscan = DBSCAN(eps=0.8, min_samples=5)
# Выполняем кластеризацию
dbscan_labels = dbscan.fit_predict(X_clusters)
# В DBSCAN метка -1 означает шум (outliers)
n_clusters_dbscan = len(set(dbscan_labels)) - (1 if -1 in dbscan_labels
else 0)
n_noise = list(dbscan_labels).count(-1)
print(f"Количество кластеров DBSCAN: {n_clusters_dbscan}")
print(f"Количество точек шума: {n_noise}")
# Подсчитываем размеры кластеров
unique_dbscan, counts_dbscan = np.unique(dbscan_labels, return_counts=True)
for cluster, count in zip(unique_dbscan, counts_dbscan):
    if cluster == -1:
        print(f"Шум: {count} точек")
    else:
        print(f"Кластер {cluster}: {count} точек")
```

Оценка и улучшение моделей

Качество модели машинного обучения оценивается различными метриками в зависимости от типа задачи. Важно понимать не только как получить результат, но и как его интерпретировать:

```
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import cross_val_score
# Для классификации: детальный отчет о качестве
print("Детальный отчет для классификации ирисов:")
print(classification_report(y_test,  y_pred_rf,  target_names=iris.target_
names))
# Матрица ошибок показывает, какие классы путает модель
conf_matrix = confusion_matrix(y_test, y_pred_rf)
print("\nМатрица ошибок:")
print(conf_matrix)
```

Кросс-валидация (cross-validation) — техника для более надежной оценки модели путем многократного разделения данных:

```
# Кросс-валидация для более надежной оценки
cv_scores = cross_val_score(rf_classifier, X, y, cv=5)
print(f"\nТочность по кросс-валидации: {cv_scores.mean():.3f} (+/- {cv_
scores.std() * 2:.3f})")
print(f"Результаты всех fold'ов: {cv_scores}")
```

Практическое применение машинного обучения требует понимания того, что выбор алгоритма зависит от природы данных, размера выборки и требований к интерпретируемости результатов. Линейные модели хорошо работают с линейно разделимыми данными и легко интерпретируются. Алгоритмы на основе деревьев (Random Forest, Gradient Boosting) эффективны для нелинейных зависимостей и смешанных типов данных. Алгоритмы кластеризации незаменимы для исследовательского анализа данных и поиска скрытых паттернов.

Машинное обучение с `scikit-learn` открывает широкие возможности для анализа данных и автоматизации принятия решений. Освоив базовые алгоритмы классификации, регрессии и кластеризации, можно переходить к более сложным техникам и специализированным областям применения.

13.4. Выбор специализации: куда двигаться дальше

Освоив основы Python и познакомившись с тремя ключевыми областями применения языка, вы стоите на пороге важного решения — выбора дальнейшего пути развития. Программирование похоже на изучение музыки: базовые навыки игры на инструменте дают вам возможность исполнять простые мелодии, но настоящее мастерство приходит с выбором стиля и глубокой специализацией в нем.

Python предоставляет уникальную возможность — войти практически в любую область современного программирования. Однако эта универсальность может стать источником растерянности. Какое направление выбрать? Как понять, что подходит именно вам? На что обратить внимание при построении карьеры?

Карта возможностей: основные направления специализации

Веб-разработка остается одной из самых востребованных областей. Если вас увлекают создание интерактивных приложений, работа с пользовательскими интерфейсами и решение задач масштабирования, это направление может стать вашим. Python предлагает мощные фреймворки — от изученного нами Flask до более комплексного Django, FastAPI для создания высокопроизводительных API.

Путь веб-разработчика предполагает изучение не только Python, но и фронтенд-технологий (HTML, CSS, JavaScript), работы с базами данных, понимания архитектурных принципов. Современный веб-разработчик должен разбираться в вопросах безопасности, производительности и пользовательского опыта.

Анализ данных и наука о данных привлекают тех, кто любит находить закономерности в хаосе информации. Если вам интересно извлекать смыслы из больших объемов данных, строить аналитические отчеты и помогать бизнесу принимать обоснованные решения, эта область откроет широкие возможности.

Специализация в области данных требует не только программистских навыков, но и понимания статистики, методов визуализации, доменных знаний в конкретной области. Современный аналитик данных работает с облачными платформами, системами распределенных вычислений, создает ETL-процессы для обработки данных.

Машинное обучение и искусственный интеллект представляют собой быстро развивающуюся область, где Python занимает лидирующие позиции. Если вас привлекает создание систем, способных обучаться и принимать решения, изучение алгоритмов распознавания образов, обработки естественного языка или компьютерного зрения — это направление может стать захватывающим путем развития.

Путь в машинном обучении требует серьезной математической подготовки, понимания алгоритмов, статистики и линейной алгебры. Современные ML-инженеры работают с большими датасетами, облачными вычислениями, создают и развертывают модели в производственных системах.

Методика самооценки: определяем свои склонности

Прежде чем выбрать специализацию, важно честно оценить свои интересы, склонности и текущие знания. Создадим простой инструмент для самоанализа:

```
def career_assessment():
    """Простая система оценки склонностей к различным областям программирования"""
    questions = {
        'web': [
            "Нравится ли вам создавать пользовательские интерфейсы?",
            "Интересно ли работать с интерактивными элементами?",
            "Привлекает ли решение задач масштабирования?",
            "Готовы ли изучать фронтенд-технологии?"
        ],
        'data': [
            "Любите ли вы работать с числами и статистикой?",
            "Интересно ли искать закономерности в данных?",
            "Нравится ли создавать визуализации и графики?",
            "Готовы ли изучать математику и статистику?"
        ],
        'ml': [
            "Привлекает ли создание 'умных' систем?",
            "Интересны ли алгоритмы и их оптимизация?",
            "Готовы ли к изучению сложной математики?",
            "Нравится ли экспериментировать с данными?"
        ]
    }

    scores = {'web': 0, 'data': 0, 'ml': 0}

    for area, area_questions in questions.items():
        print(f"\n--- Вопросы по направлению: {area.upper()} ---")
        for question in area_questions:
            while True:
                answer = input(f"{question} (да/нет): ").lower().strip()
                if answer in ['да', 'нет', 'yes', 'no']:
                    if answer in ['да', 'yes']:
                        scores[area] += 1
                    break
                print("Пожалуйста, ответьте 'да' или 'нет'")

    return scores
def interpret_results(scores):
    """Интерпретация результатов оценки"""
    max_score = max(scores.values())
```

```
    if max_score == 0:
         return "Возможно, стоит больше времени потратить на изучение раз-
личных областей"

     top_areas = [area for area, score in scores.items() if score == max_
score]

    area_names = {
        'web': 'Веб-разработка',
        'data': 'Анализ данных',
        'ml': 'Машинное обучение'
    }

    if len(top_areas) == 1:
       return f"Наиболее подходящее направление: {area_names[top_areas[0]]}"
    else:
        areas_str = ", ".join([area_names[area] for area in top_areas])
        return f"У вас равные склонности к: {areas_str}"
```

Этот инструмент поможет вам структурированно подумать о своих предпочтениях. Помните, что результаты носят рекомендательный характер — окончательное решение всегда остается за вами.

Построение траектории обучения

Независимо от выбранного направления эффективное развитие следует определенным принципам. **Принцип постепенности** предполагает, что сложные навыки строятся на основе простых. Не пытайтесь сразу браться за самые сложные проекты — двигайтесь пошагово, закрепляя каждый уровень знаний.

Принцип практической направленности означает, что теоретические знания должны немедленно подкрепляться практикой. Создавайте проекты, решайте реальные задачи, участвуйте в открытых проектах на GitHub.

Принцип разнообразия источников предполагает использование различных способов обучения: книги, онлайн-курсы, видеолекции, подкасты, конференции, общение с коллегами. Каждый источник дает уникальную перспективу понимания материала.

Для каждого направления существуют свои специфические ресурсы и траектории развития.

Веб-разработка: от Flask к полнофункциональным приложениям

После изучения основ Flask логичным следующим шагом станет освоение Django — более комплексного фреймворка, предоставляющего готовые решения для аутентификации, администрирования, ORM и многого другого. Django следует принципу «батарейки включены» (batteries included), что значительно ускоряет разработку.

Современная веб-разработка требует понимания фронтенд-технологий. Изучите основы HTML5, CSS3, освойте JavaScript и один из современных фреймворков — React, Vue.js или Angular. Понимание того, как работает клиентская часть приложения, сделает вас более эффективным разработчиком.

Обязательно изучите работу с базами данных. Начните с реляционных БД (PostgreSQL, MySQL), освойте SQL, понимание принципов нормализации. Позднее познакомьтесь с NoSQL-решениями — MongoDB, Redis.

Изучите принципы REST API, освойте FastAPI для создания высокопроизводительных веб-сервисов. Познакомьтесь с GraphQL как альтернативным подходом к проектированию API.

Обратите внимание на DevOps-практики: контейнеризация с Docker, автоматизация развертывания, мониторинг приложений, работа с облачными платформами (AWS, Google Cloud, Azure).

Анализ данных: от `pandas` к экосистеме больших данных

Углубьте знания библиотек для анализа данных. Изучите продвинутые возможности `pandas`: группировка, агрегация, работа с временными рядами, оптимизация производительности при работе с большими датасетами.

Освойте продвинутые инструменты визуализации: `Seaborn` для статистических графиков, `Plotly` для интерактивных визуализаций, `Bokeh` для веб-приложений с графиками.

Изучите статистические методы и их реализацию в Python. Библиотека SciPy предоставляет широкий набор статистических функций, а `statsmodels` — специализированные инструменты для статистического моделирования.

Познакомьтесь с инструментами для работы с большими данными: Apache Spark (PySpark), Dask для параллельных вычислений, работа с колоночными форматами данных (Parquet, Apache Arrow).

Изучите облачные решения для анализа данных: Google BigQuery, Amazon Redshift, Microsoft Azure Analytics, понимание принципов хранилищ данных (data warehouse) и озер данных (data lake).

Машинное обучение: от `scikit-learn` к глубокому обучению

Углубьте математические основы: линейная алгебра, статистика, теория вероятностей. Эти знания критически важны для понимания того, как работают алгоритмы машинного обучения.

Изучите продвинутые алгоритмы машинного обучения: ансамблевые методы (Random Forest, Gradient Boosting), методы снижения размерности (PCA, t-SNE), алгоритмы кластеризации.

Освойте глубокое обучение с помощью фреймворков TensorFlow или PyTorch. Изучите основы нейронных сетей, сверточные сети для компьютерного зрения, рекуррентные сети для обработки последовательностей.

Изучите специализированные области: обработка естественного языка (NLTK, spaCy, transformers), компьютерное зрение (OpenCV, PIL), обработка сигналов.

Важная часть работы ML-инженера — развертывание моделей в продакшен. Изучите MLOps-практики: версионирование моделей, мониторинг производительности, A/B тестирование моделей.

Построение профессионального портфолио

Независимо от выбранной специализации, ключевым элементом карьерного роста становится создание портфолио проектов. Портфолио должно демонстрировать не только технические навыки, но и способность решать реальные задачи:

```
def create_portfolio_checklist():
    """Чек-лист для создания профессионального портфолио"""
```

```
portfolio_items = {
    'web': [
        "Полнофункциональное веб-приложение с аутентификацией",
        "REST API с документацией",
        "Интеграция с внешними сервисами",
        "Развернутый проект на хостинге"
    ],
    'data': [
        "Исследовательский анализ данных (EDA)",
        "Дашборд с интерактивными графиками",
        "Автоматизированный отчет",
        "Проект прогнозирования на временных рядах"
    ],
    'ml': [
        "Проект классификации с оценкой качества",
        "Система рекомендаций",
        "Обработка естественного языка или компьютерное зрение",
        "Развернутая ML-модель как API"
    ]
}

print("Рекомендуемые проекты для портфолио:")
for area, projects in portfolio_items.items():
    print(f"\n{area.upper()}:")
    for i, project in enumerate(projects, 1):
        print(f"  {i}. {project}")

general_tips = [
    "Код должен быть чистым и документированным",
    "README-файл с описанием проекта и инструкциями запуска",
    "Использование системы контроля версий (Git)",
    "Демонстрация результатов (скриншоты, графики, метрики)"
]

print("\nОбщие рекомендации:")
for tip in general_tips:
    print(f"  • {tip}")
```

Помните: работодатели и клиенты оценивают не количество проектов в портфолио, а их качество и релевантность. Лучше иметь три-четыре хорошо выполненных проекта, чем десяток поверхностных.

Развитие soft skills и профессиональных навыков

Техническая экспертиза — это лишь часть успешной карьеры в программировании. Развивайте навыки коммуникации: умение объяснять сложные технические концепции простым языком, способность работать в команде, навыки презентации результатов работы.

Изучайте методологии разработки: Agile, Scrum, понимание жизненного цикла разработки ПО. Эти знания помогут вам эффективно работать в команде и понимать контекст своей работы.

Развивайте навыки решения проблем: способность разбивать сложные задачи на простые, систематический подход к отладке, умение находить информацию и учиться самостоятельно.

Участвуйте в профессиональном сообществе: посещайте конференции и митапы, участвуйте в open source проектах, ведите технический блог или канал, делитесь знаниями с другими разработчиками.

Заключение

Путь, который мы прошли вместе

Закрывая последнюю страницу этой книги, вы завершаете удивительное путешествие — от первой команды `print("Привет, мир!")` до создания сложных объектно-ориентированных систем, от простейших арифметических операций до работы с машинным обучением и веб-сервисами. Этот путь превратил вас из человека, который лишь пользовался компьютером, в того, кто может заставить его думать и работать по собственным алгоритмам.

Оглядываясь назад, можно увидеть, как естественно и логично выстраивалась наша учебная траектория. **Первые главы** заложили фундамент — мы изучили основы взаимодействия с Python, освоили типы данных и научились мыслить в терминах переменных и операций. Уже тогда вы начали понимать, что программирование — это не магия, а строгая система правил и принципов.

Главы в середине книги научили нас управлять логикой программ. Условные конструкции показали, как программы принимают решения, циклы продемонстрировали силу автоматизации, а функции открыли красоту модульного программирования. В этих главах сформировались ваши алгоритмические навыки — способность разбивать сложную задачу на простые шаги и элегантно организовывать код.

Заключительные главы превратили вас в настоящего разработчика. Структуры данных научили эффективно организовывать информацию, объектно-ориентированное программирование показало, как моделировать реальный мир в коде, а работа с внешними данными и современными технологиями открыла безграничные возможности применения ваших навыков.

Python как первый, но не последний язык

Выбор Python в качестве первого языка программирования был неслучайным. Его философия простоты и читаемости позволила сосредоточиться на изучении фундаментальных концепций программирования, не отвлекаясь на сложности синтаксиса. Python научил вас думать «пайтонически» — писать код, который не просто работает, но является ясным, элегантным и эффективным.

Однако важно понимать, что Python — это инструмент, хотя и очень мощный. **Истинная ценность ваших знаний** заключается не во владении конкретным синтаксисом, а в понимании алгоритмов, структур данных, принципов проектирования и архитектуры программ. Эти знания легко переносятся между языками программирования.

Если в будущем вам потребуется изучить JavaScript для веб-разработки, Java для enterprise-приложений, C++ для системного программирования или Go

для распределенных систем, вы обнаружите, что основные концепции остаются теми же. Меняется лишь способ их выражения в коде.

Влияние программирования на мышление

Освоение программирования меняет способ мышления человека. Вы начинаете видеть алгоритмы в повседневной жизни, замечаете возможности автоматизации, мыслите более структурно и системно. Это **вычислительное мышление** становится инструментом решения проблем далеко за пределами программирования.

Программирование развивает терпение и настойчивость — качества, необходимые для отладки сложных багов и оптимизации производительности. Оно учит точности — компьютер выполняет именно то, что вы написали, а не то, что имели в виду. Оно прививает привычку к непрерывному обучению — технологии меняются быстро, и только постоянное развитие позволяет оставаться актуальным специалистом.

Ответственность разработчика

С освоением программирования приходит и ответственность. Программы, которые мы создаем, влияют на жизни людей. Банковские системы управляют финансами, медицинские приложения помогают диагностировать заболевания, социальные сети формируют общественное мнение.

Этический аспект программирования становится все более важным. Вопросы приватности данных, алгоритмической справедливости, безопасности информации — это не абстрактные философские проблемы, а практические задачи, с которыми сталкивается каждый разработчик.

Помните о важности создания **инклюзивного и доступного** программного обеспечения. Ваши программы должны работать для людей с различными возможностями, культурными особенностями, техническими ограничениями. Хороший код — это не только эффективный код, но и код, который служит всем людям.

Заключительные размышления

Программирование — это одна из немногих профессий, где ваши идеи могут мгновенно материализоваться в работающие приложения. Где вы можете создать что-то, чего раньше не существовало, и поделиться этим с миллионами людей. Где единственным ограничением является ваше воображение и готовность учиться.

Вы вступаете в мир, где **творчество сочетается с логикой**, где **искусство встречается с наукой**. Мир, где небольшая команда разработчиков может изменить целые индустрии, где стартап в гараже может стать глобальной корпорацией, где открытый исходный код позволяет людям со всего мира совместно создавать удивительные вещи.

Дорога, которую вы прошли с этой книгой, — это лишь первые шаги в удивительном путешествии. Впереди вас ждут новые языки программирования, фреймворки, парадигмы, технологии. Каждый новый проект будет учить вас чему-то новому, каждая решенная задача будет делать вас более опытным разработчиком.

Программирование — это не только профессия, это способ мышления, инструмент творчества и средство решения проблем. Навыки, которые вы получили, будут служить вам всю жизнь и откроют двери в самые разные области человеческой деятельности.

Добро пожаловать в удивительный мир программирования. Ваше путешествие только начинается, и лучшие открытия еще впереди.

16+

Справочное издание
Анықтамалық басылым

Серия «Программирование от А до С++»

КЛИМОВИЧ Александр Дмитриевич

ПРАВИЛЬНЫЙ СПОСОБ ИЗУЧИТЬ PYTHON С ПРИМЕРАМИ И ЗАДАЧАМИ

Ответственный за выпуск: *И. Резько*
Ответственный редактор: *А. Ходякова*
Оформление обложки: *С. Дарсич*

Подписано в печать 15.02.2026.
Формат 84x108/16 . Бумага офсетная. Печать офсетная.
Гарнитура Noto Serif SemiCondensed.
Усл. печ. л. 36,4. Тираж 2000 экз. Заказ №

Общероссийский классификатор продукции ОК-034-2014 (КПЕС 2008);
58.11.1 — книги, брошюры печатные.

Изготовлено в 2026 г.
Произведено в Российской Федерации.
Изготовитель: ООО «Издательство АСТ».
129085, Российская Федерация, г. Москва, Звёздный бульвар, дом 21,
строение 1, комната 705, пом. I, 7 этаж.
Адрес места осуществления деятельности по изготовлению продукции:
123112, Российская Федерация, г. Москва, Пресненская набережная, д. 6, стр. 2,
Деловой комплекс «Империя», 14, 15 этаж.
Наш электронный адрес: ask@ast.ru
Наш сайт: www.ast.ru
Интернет-магазин: www.book24.ru

Өндіруші: «Издательство АСТ» ЖШҚ 129085, Ресей Федерациясы,
Мәскеу қ., Звёздный бульвары, 21-үй, 1-құрылыс, 705-бөлме, I үй-жай, 7-қабат.
Өнім өндіру қызметін жүзеге асыру мекен-жайы:
123112, Ресей Федерациясы, Мәскеу қ.,
Пресненская жағ., 6-үй, 2-құр., «Империя» іскерлік кешені, 14, 15-қабат.
Біздің электрондық мекенжайымыз: www.ast.ru E-mail: ask@ast.ru
Интернет-магазин: www.book24.ru Интернет-дүкен: www.book24.kz
Импортер в Республику Казахстан, дистрибьютор и представитель по приёму претензий
на продукцию в Республике Казахстан: ТОО «РДЦ-Алматы».
г. Алматы, ул. Домбровского, 3«а», литер Б, офис 1.
Қазақстан Республикасына импорттаушы дистрибьютор және Қазақстан Республикасында
өнімге шағымдар қабылдау жөніндегі өкіл: «РДЦ-Алматы» ЖШС.
Алматы қ., Домбровский көш., 3«а», Б литері, офис 1.
Тел.: 8(727) 2 51 59 90,91, факс: 8 (727) 251 59 92 ішкі 107;
E-mail: RDC-Almaty@eksmo.kz, www.book24.kz
Өндірілген жылы: 2026. Өнімнің жарамдылық мерзімі шектелмеген.
Сертификаттауға жатпайды.
Ресей Федерациясында өндірілген.

Перед вами практическое руководство по программированию на одном из **самых востребованных сегодня языков — Python**, которое формирует не просто знание синтаксиса, а системное мышление разработчика, способного решать сложные задачи в любой области.

Книга строится на принципе глубокого понимания вместо механического запоминания. Каждая концепция раскрывается через трехмерную модель: что представляет собой конструкция, почему она работает именно так и когда ее следует применять. **От первых команд и типов данных до объектно-ориентированного программирования и работы с современными технологиями** — читатель проходит путь от обычного пользователя компьютера до создателя реальных программных воплощений.

Особое внимание уделено **культуре качественного кода**: правильные практики именования, структурирования, документирования и тестирования интегрированы в каждый раздел с самого начала. Книга развивает алгоритмическое мышление — умение декомпозировать сложные проблемы, создавать абстракции и находить элегантные решения.

Автор рассматривает Python не как самоцель, а как инструмент формирования вычислительного мышления, которое применимо далеко за пределами сферы программирования. В книге также затрагиваются этические аспекты разработки и ответственность программиста в создании инклюзивного и безопасного программного продукта.

Данное издание предназначено для всех, кто хочет **освоить программирование на качественном уровне** — от абсолютных новичков до тех, кто стремится систематизировать свои знания и развить профессиональное мышление разработчика.

КНИГИ ДЛЯ ЛЮБОГО НАСТРОЕНИЯ ЗДЕСЬ:
www.ast.ru / www.book24.ru
vk.com/izdatelstvoast
ok.ru/izdatelstvoast

ИЗДАТЕЛЬСКАЯ ГРУППА АСТ

ISBN 978-5-17-178635-9
9 785171 786359