PYTHON

Виталий Шефер

От первой строки до своих проектов



ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Виталий Шефер

Python

«Автор»
2026

Шефер В.

Python / В. Шефер — «Автор», 2026

Практическое руководство по Python для тех, кто начинает с нуля: от первой строки кода до восьми собственных проектов.Книга написана для Python 3.15 и Windows, отличия для macOS и Linux даны во врезках. Сначала вы освоите основы: переменные, условия, циклы, списки, словари, функции, файлы и классы. Потом - инструменты программиста: тесты, Git, базы данных SQLite, работу с интернетом и с нейросетью. Затем шаг за шагом напишете игру "Змейка", менеджер паролей, трекер привычек, семейный бюджет с графиками, программу, которая следит за погодой и ценами, Telegram-бота для записи клиентов и сайт с рецептами. В последней части пройдёте путь от идеи до готовой программы - мини-CRM для самозанятого. Примеры бытовые и с юмором: кот Барсик, дача, семейные расходы. Весь код проверен запуском.

© Шефер В., 2026
© Автор, 2026

Содержание

Введение. Как читать эту книгу 16
- Для кого эта книга 17
- Что вы сделаете к концу книги 18
- Как устроена книга 19
- Как устроены главы 20
  - Задания "Попробуйте сами" 20
  - Главы-проекты 21
  - Итоги 21
  - Врезки 21
  - Термины 21
- Почему Python 3.15 22
- Код примеров 23
- Что делать, если не получается 24
- Как читать, чтобы был толк 25

Глава 1. Первая программа 26
- Что такое программа 27
- Почему Python 28
- Устанавливаем Python 3.15 29
  - Установка на Windows 29
- Интерактивный режим 30
- Редактор кода 32
  - Visual Studio Code 32
  - IDLE 32
- Первая программа 33
  - Несколько строк 33
  - Картинка из символов 34
  - Кавычки внутри текста 34
- Как читать сообщения об ошибках 36
  - Синтаксические ошибки 36
  - Ещё два частых промаха 37
  - Маленький словарик ошибок 38
- Итоги 40

Глава 2. Переменные и типы данных 41
- Переменные 42
  - Как называть переменные 42
- Числа 44
  - Арифметика 44
  - Целое деление и остаток 46
  - Сокращённые записи 46
  - Сюрприз с дробными числами 47
- Строки 48
  - Особые символы 48
  - Многострочный текст 49
  - f-строки 49
- Логический тип 51
- Ввод с клавиатуры 52

Ввод - всегда строка 52
Кошачьи годы 54
Комментарии 55
Собираем всё вместе: поездка на дачу 56
Итоги 58
Глава 3. Условия 59
Если: инструкция if 60
Иначе: else 62
Сравнения 63
Несколько вариантов: elif 65
Логические операторы: and, or, not 68
Что Python считает истиной 69
Частые ошибки в условиях 71
Вложенные условия 73
Условие в одну строку 74
Выбор по образцу: match 75
Собираем всё вместе: викторина 77
Итоги 79
Глава 4. Циклы 80
Цикл while 81
Цикл с вопросами 81
Копим на велосипед 82
Цикл for и функция range 84
Таблица умножения 85
Накопление суммы 86
Какой цикл выбрать 87
Выход из цикла: break и continue 89
Цикл с else 90
Вложенные циклы 92
Игра "Угадай число" 94
Ограничим попытки 95
Поменяемся ролями 96
Итоги 99
Глава 5. Списки и кортежи 100
Что такое список 101
Индексы 102
Срезы 104
Изменение списка 106
Два имени - один список 107
Сортировка и статистика 109
Перебор списка 111
Список, который растёт в цикле 112
Генераторы списков 114
Распаковка в генераторах 115
Кортежи 116
Список кортежей 117
Собираем всё вместе: викторина на списках 119
Итоги 121
Глава 6. Словари и множества 122

Что такое словарь 123
Если ключа нет 124
Что может быть ключом 125
Перебор словаря 126
Подсчёт с помощью словаря 127
Вложенные структуры 129
Список словарей 129
Словарь списков 130
Словарь словарей 131
Множества 132
Убрать повторы 132
Операции над множествами 133
frozendict - новинка Python 3.15 134
frozendict как ключ 135
Собираем всё вместе: записная книжка 137
Итоги 140
Глава 7. Строки и текст 141
Методы строк 142
Ответ "да" в любом виде 143
Поиск и замена 145
Приводим телефоны к одному виду 145
Разбиение и склейка 147
Форматирование чисел 149
Выравнивание и таблицы 150
Даты 152
Даты по-русски 152
Русский текст: регистр, сортировка и буква ё 155
Поиск, который не замечает ё 156
Собираем всё вместе: разбор отзывов 157
Итоги 159
Глава 8. Функции 160
Зачем нужны функции 161
Объявление и вызов 162
Параметры и аргументы 163
Возвращаемое значение 165
print или return 165
Несколько значений и ранний выход 166
Области видимости 168
Произвольное число аргументов 171
Документация и аннотации типов 173
Короткие функции: lambda 175
Собираем всё вместе: викторина на функциях 177
Итоги 180
Глава 9. Ошибки и отладка 181
Три вида ошибок 182
Как читать трассировку 183
Частые исключения 184
Перехват исключений: try и except 186
Несколько видов исключений 186

Блоки else и finally 188
Свои исключения 190
Отладка 192
Отладка выводом 192
Отладчик 192
Отладчик pdb 193
Логирование 195
Собираем всё вместе: надёжный расчёт поездки 197
Итоги 199
Глава 10. Файлы 200
Запись и чтение 201
Дописать в конец 202
Кодировки и UTF-8 по умолчанию 204
Пути к файлам: модуль pathlib 206
Наводим порядок в загрузках 207
CSV: таблицы в тексте 210
JSON: данные со структурой 212
Что выбрать: TXT, CSV или JSON 216
Итоги 217
Глава 11. Модули и пакеты 218
Модули и import 219
Зачем if __name__ == "__main__" 220
Пакеты 222
Стандартная библиотека 224
Конец ознакомительного фрагмента. 225

Виталий Шефер
Python

Python: от первой строки до своих проектов
Практическое руководство

Виталий Шефер

Оглавление
Введение. Как читать эту книгу
Для кого эта книга
Что вы сделаете к концу книги
Как устроена книга
Как устроены главы
Почему Python 3.15
Код примеров
Что делать, если не получается
Как читать, чтобы был толк
Глава 1. Первая программа
Что такое программа
Почему Python
Устанавливаем Python 3.15
Интерактивный режим
Редактор кода
Первая программа
Как читать сообщения об ошибках
Итоги
Глава 2. Переменные и типы данных
Переменные
Числа
Строки
Логический тип
Ввод с клавиатуры
Комментарии
Собираем всё вместе: поездка на дачу
Итоги
Глава 3. Условия
Если: инструкция if
Иначе: else
Сравнения
Несколько вариантов: elif
Логические операторы: and, or, not
Частые ошибки в условиях
Вложенные условия
Условие в одну строку
Выбор по образцу: match
Собираем всё вместе: викторина

Итоги
Глава 4. Циклы
Цикл while
Цикл for и функция range
Выход из цикла: break и continue
Вложенные циклы
Игра "Угадай число"
Итоги
Глава 5. Списки и кортежи
Что такое список
Индексы
Срезы
Изменение списка
Сортировка и статистика
Перебор списка
Генераторы списков
Кортежи
Собираем всё вместе: викторина на списках
Итоги
Глава 6. Словари и множества
Что такое словарь
Перебор словаря
Вложенные структуры
Множества
frozendict - новинка Python 3.15
Собираем всё вместе: записная книжка
Итоги
Глава 7. Строки и текст
Методы строк
Поиск и замена
Разбиение и склейка
Форматирование чисел
Даты
Русский текст: регистр, сортировка и буква ё
Собираем всё вместе: разбор отзывов
Итоги
Глава 8. Функции
Зачем нужны функции
Объявление и вызов
Параметры и аргументы
Возвращаемое значение
Области видимости
Произвольное число аргументов
Документация и аннотации типов
Короткие функции: lambda
Собираем всё вместе: викторина на функциях
Итоги
Глава 9. Ошибки и отладка
Три вида ошибок

Как читать трассировку
Перехват исключений: try и except
Блоки else и finally
Свои исключения
Отладка
Логирование
Собираем всё вместе: надёжный расчёт поездки
Итоги
Глава 10. Файлы
Запись и чтение
Кодировки и UTF-8 по умолчанию
Пути к файлам: модуль pathlib
CSV: таблицы в тексте
JSON: данные со структурой
Что выбрать: TXT, CSV или JSON
Итоги
Глава 11. Модули и пакеты
Модули и import
Пакеты
Стандартная библиотека
Сторонние библиотеки и pip
Виртуальные окружения
uv: быстрый помощник
Итоги
Глава 12. Классы и объекты
Класс и объект
Методы, которые меняют объект
Корзина покупок
Наследование
Классы данных: dataclasses
Когда нужны классы
Собираем всё вместе: копилка Димы
Итоги
Глава 13. Тесты
Зачем нужны тесты
Проверка через assert
pytest
Тест до кода
Тесты для классов
Что и как тестировать
Итоги
Глава 14. Чистый код и Git
Код пишут для людей
Понятные имена
ruff: оформление проверяет программа
Структура проекта
Git: машина времени для кода
Ветки
GitHub и российские площадки

Итоги
Глава 15. Базы данных SQLite
Таблицы, строки и столбцы
SQL с нуля
Модуль sqlite3
Связи между таблицами
Итоги
Глава 16. Интернет и API
Как устроен интернет-запрос
Библиотека requests
JSON и API
Когда что-то идёт не так
POST: отправляем данные
Ключи доступа и где их хранить
Итоги
Глава 17. Программирование вместе с нейросетью
Что умеют ИИ-помощники и где ошибаются
Как спрашивать
Как проверять ответ
Пароли и личные данные не отправлять
Как не разучиться думать самому
Итоги
Глава 18. Змейка: окно и управление
Готовим проект
Первое окно
Координаты и рисование
Поле в клеточку
Змейка ползёт
Управление стрелками
Змейка - объект
Что можно улучшить
Итоги
Глава 19. Змейка: еда, очки и рекорды
Змейка растёт и врезается
Правила: еда, очки и уровни
Таблица рекордов в файле
Звуки, сочинённые кодом
Меню и экраны игры
Что можно улучшить
Итоги
Глава 20. Менеджер паролей
Что будем делать
Надёжный пароль
Как шифровать
Хранилище
Первое окно на tkinter
Окно "Сейфа"
Поиск и буфер обмена
Проверяем "Сейф"

Что можно улучшить
Итоги
Глава 21. Трекер полезных привычек
Что будем делать
База привычек
Серии и статистика
Окно с сеткой дней
Программа в один файл .exe
Что можно улучшить
Итоги
Глава 22. Семейный бюджет: учёт доходов и расходов
Что будем делать
Деньги в копейках
База бюджета
Команды в командной строке
Выписка из банка
Кто сколько тратит
Тесты
Что можно улучшить
Итоги
Глава 23. Семейный бюджет: отчёты и графики
Готовим данные
Первое знакомство с pandas
Итоги месяца и сравнение месяцев
Графики matplotlib
Отчёт в Excel
Тесты
Что можно улучшить
Итоги
Глава 24. Погода и цены: программа-наблюдатель
Что будем делать
Прогноз на завтра
Цена со страницы магазина
Уведомление в Telegram
Наблюдатель целиком
Расписание проверок
Тесты без интернета
Что можно улучшить
Итоги
Глава 25. Telegram-бот записи на маникюр: первые шаги
Что будем делать
Как бот разговаривает с Telegram
Бот у BotFather
Асинхронность: пока сохнет лак
Первый бот
Бот студии
Тесты без Telegram
Где запускать бота
Что можно улучшить

Итоги
Глава 26. Telegram-бот записи на маникюр: запись и напоминания
Что будем делать
Как хранить записи
Правила записи
Тесты правил записи
Тексты и кнопки
Бот: запись по шагам
Мои записи: перенос и отмена
Расписание мастера
Напоминания накануне
Бот может переехать
Что можно улучшить
Итоги
Глава 27. Сайт с рецептами: первые страницы
Что будем делать
Как работает сайт
Первая страница
Рецепты в базе: многие-ко-многим
Шаблоны
Сайт целиком
Безопасность: экранирование
Тесты сайта
Что можно улучшить
Итоги
Глава 28. Сайт с рецептами: добавление, поиск и выкладка
Что будем делать
Поиск без заглавных букв и без ё
Фото рецепта
Вход для автора
Форма нового рецепта
Проверяем в браузере
Тесты
Выкладка в интернет
Что можно улучшить
Итоги
Глава 29. От идеи к плану
Разговор с заказчиком
Что нужно знать о налоге самозанятых
Требования: истории пользователя
Чем будем делать
Список задач
Модель данных
Эскиз окна
План версий
Как будем проверять
Папка и репозиторий проекта
Что можно улучшить
Итоги

Глава 30. Мини-CRM: клиенты и заказы
Что будем делать
База
Напоминания
Проверка форм
Тесты
Окно
Приёмка
Что можно улучшить
Итоги
Глава 31. Мини-CRM: документы, отчёты и налог
Что будем делать
Налог за месяц
База версии 0.3
Настройки
Счёт и акт в Word
Отчёт в Excel
Резервная копия
Окно версии 0.3
Тесты
Выпуск версии 0.3
Что можно улучшить
Итоги
Глава 32. Готовый продукт и что дальше
Что будем делать
Где программа хранит данные
Окно версии 1.0
Настройки по образцу
Сборка .exe
Проверка перед выпуском
README, requirements.txt и .gitignore
Выпуск версии 1.0
Публикация кода
Как выпускать следующие версии
Портфолио
Куда расти
Что можно улучшить
Итоги
Приложение А. Ответы к заданиям "Попробуйте сами"
Глава 1. Первая программа
Глава 2. Переменные и типы данных
Глава 3. Условия
Глава 4. Циклы
Глава 5. Списки и кортежи
Глава 6. Словари и множества
Глава 7. Строки и текст
Глава 8. Функции
Глава 9. Ошибки и отладка
Глава 10. Файлы

Глава 11. Модули и пакеты
Глава 12. Классы и объекты
Глава 13. Тесты
Глава 14. Чистый код и Git
Глава 15. Базы данных SQLite
Глава 16. Интернет и API
Глава 17. Программирование вместе с нейросетью
Приложение Б. Шпаргалка по Python
Запуск и установка (главы 1, 11)
Переменные, типы, ввод и вывод (главы 2, 3)
Числа и операции (глава 2)
Форматирование строк (главы 2, 7)
Сравнения и логика (глава 3)
Условия (глава 3)
Циклы (главы 4, 5)
Строки (глава 7)
Списки и кортежи (глава 5)
Словари и множества (глава 6)
Функции (глава 8)
Ошибки и отладка (глава 9)
Файлы (глава 10)
Модули и стандартная библиотека (глава 11)
Классы (глава 12)
Тесты (глава 13)
SQLite и SQL (глава 15)
Интернет и API (глава 16)
Git (глава 14)
Новое в Python 3.15
Приложение В. Частые ошибки новичков и как их исправить
Как читать сообщение об ошибке
Ошибки, которые видны до запуска
Ошибки во время работы
Ошибки без сообщения
Ошибки окружения в Windows
Если ничего не помогает

Введение. Как читать эту книгу

Однажды кот Барсик прошёлся по клавиатуре, пока хозяйка отвернулась за чаем. На экране появилось "пппппппппп7777ъ". Барсик гордо сел рядом и сделал вид, что так и задумано. Программой это, к сожалению, не было: компьютер не понял, чего от него хотят, и ничего не сделал.

Эта книга о том, как сделать наоборот: написать такие строки, которые компьютер поймёт и выполнит. Посчитать семейные расходы за месяц, напомнить клиенту о записи на маникюр, сообщить, что любимая кофеварка подешевела, нарисовать змейку, которая гоняется за яблоками. Всё это мы будем делать на языке Python (произносится "пайтон", в России часто говорят "питон", и это тоже правильно).

Для кого эта книга

Для тех, кто начинает с нуля. Не нужно знать, что такое переменная, не нужно помнить школьную алгебру дальше умножения и процентов. Нужны компьютер, желание и немного упрямства: первые ошибки бывают у всех, и это нормальная часть работы.

Книга написана для взрослых и для подростков от 14 лет. Кто-то хочет автоматизировать скучную работу, кто-то присматривается к профессии программиста, кто-то хочет написать игру или бота для мамы, которая делает маникюр на дому. Всем подойдёт один и тот же путь: сначала основы, потом инструменты, потом настоящие проекты.

Основная система в книге - Windows: у большинства читателей стоит именно она. Если у вас macOS или Linux, ничего страшного. Python везде одинаковый, а отличия (установка, команды в терминале, пути к файлам) вынесены во врезки "Если у вас macOS или Linux".

Что вы сделаете к концу книги

Вот программы, которые вы напишете своими руками:

игру "Змейка" с едой, уровнями, звуком и таблицей рекордов;

менеджер паролей с мастер-паролем и шифрованием;

трекер полезных привычек с окном и сеткой дней, собранный в обычную программу для Windows (файл .exe);

семейный бюджет: учёт доходов и расходов, импорт банковской выписки, отчёты, графики и таблица Excel;

программу-наблюдателя: она смотрит прогноз погоды и цену товара в магазине и пишет вам в Telegram, когда цена упала или обещают дождь;

Telegram-бота, который записывает клиентов на маникюр, показывает свободные окошки и напоминает о визите накануне;

сайт с рецептами: список, поиск по ингредиентам, добавление рецептов с фото и выкладка в интернет;

мини-CRM для самозанятого: клиенты, заказы, напоминания, счета и акты в Word, отчёты в Excel и расчёт налога.

Последний проект особенный. На нём мы пройдём весь путь своего проекта: от идеи и списка задач до готовой программы с описанием и публикацией кода. Этот путь потом можно повторить с любой своей идеей.

Как устроена книга

Книга состоит из четырёх частей.

Часть 1. Основы (главы 1-12). Установка Python, первая программа, переменные, условия, циклы, списки, словари, строки, функции, ошибки, файлы, модули и классы. Это фундамент. Главы этой части лучше читать строго по порядку: каждая опирается на предыдущие.

Часть 2. Инструменты программиста (главы 13-17). Тесты, чистый код, система контроля версий Git, базы данных SQLite, работа с интернетом и программирование вместе с нейросетью. Без этого можно написать программу, но трудно написать хорошую программу.

Часть 3. Проекты (главы 18-28). Семь проектов из списка выше, все, кроме мини-CRM. Каждому отведена одна или две главы. Проекты можно читать в любом порядке, когда за плечами первые две части. Если проекту нужна тема из основ, в тексте есть ссылка на главу, где она разобрана.

Часть 4. Свой проект (главы 29-32). Мини-CRM для самозанятого кондитера Марины: от разговора "что тебе нужно?" до готовой программы, которую можно отдать заказчику.

В конце книги три приложения: ответы к заданиям, шпаргалка по Python и разбор частых ошибок новичков.

Как устроены главы

Каждая глава объясняет новое на бытовых примерах: магазин, дача, семейный бюджет, кот. Объяснение сразу подкрепляется кодом. Программы в книге называются листингами (от английского listing, "распечатка"), у каждого есть номер и имя файла. Например, вот листинг из этого введения. Пока не пытайтесь разобраться, как он устроен: к концу второй главы вы напишете такой же сами.

Листинг 0.1. barsik.py

```
name = input("Как вас зовут? ")

print(f"Приятно познакомиться, {name}! Меня зовут Барсик.")

print("Я кот. Я умею мурлыкать и ронять вещи со стола.")
```

Номер 0.1 значит "введение, первый листинг". В главе 5 листинги будут называться 5.1, 5.2 и так далее. Под листингом обычно показано, что программа выводит на экран:

```
Как вас зовут? Маша

Приятно познакомиться, Маша! Меня зовут Барсик.

Я кот. Я умею мурлыкать и ронять вещи со стола.
```

Здесь важная деталь. Программа спросила имя, а человек ответил "Маша" и нажал Enter. В книге то, что вводит человек, показано прямо в выводе, после вопроса программы, ровно так, как это выглядит на экране при запуске. Весь вывод в книге получен настоящим запуском программ на Python 3.15, а не придуман.

Иногда в книге встречается и другой вид кода: короткие пробы в интерактивном режиме Python. Строки, которые вводит человек, начинаются с трёх угловых скобок >>>, а ответ Python идёт следом без скобок:

```
>>> 2 + 2

4

>>> "мяу" * 3

'мяумяумяу'
```

Что такое интерактивный режим и как в него попасть, расскажем в первой главе.

Задания "Попробуйте сами"

В главах 1-17 по ходу текста встречаются задания. Они выглядят так:

Попробуйте сами. Измените программу barsik.py так, чтобы кот рассказывал о себе ещё одну вещь.

В настоящих главах у заданий есть номер, например "Попробуйте сами 4.3". Ответы ко всем заданиям собраны в приложении А. Совет простой и проверенный: сначала решите сами, даже если решение получится неуклюжим, и только потом загляните в ответ. Программированию учатся руками, а не глазами. Пятнадцать минут собственных мучений над задачей дают больше, чем час чтения готовых решений.

Почти у любой задачи есть несколько правильных решений. Если ваша программа делает то, что требуется, а ответ в приложении написан иначе, это не ошибка. Сравните два варианта и подумайте, какой понятнее.

Главы-проекты

В главах 18-32 заданий с номерами нет: там вся глава - одно большое задание. Вместо них в конце каждой такой главы есть раздел "Что можно улучшить" с идеями, куда развивать программу дальше. Эти идеи без готовых ответов, как в настоящей работе.

Итоги

Каждая глава заканчивается разделом "Итоги": пять-восемь пунктов о самом главном. Если через месяц понадобится вспомнить, что было в главе про словари, начните с итогов.

Врезки

Некоторые абзацы выделены и начинаются с жирного заголовка:

Если у вас macOS или Linux. Здесь написано, чем отличаются действия на этих системах.

Новое в Python 3.15. Здесь рассказано о новинках свежей версии, которые полезны новичку.

Частая ошибка. Здесь разобраны грабли, на которые наступает почти каждый.

Совет. Здесь полезные приёмы, без которых можно обойтись, но с ними проще.

Термины

Программисты говорят на смеси русского и английского. Когда в книге появляется новый термин, рядом в скобках стоит его английский оригинал, например: переменная (variable), функция (function), список (list). Английские слова пригодятся, когда вы будете искать ответы в интернете и читать сообщения об ошибках: Python пишет их по-английски.

Почему Python 3.15

Python 3.15 вышел 9 октября 2026 года. Книга написана и проверена именно на нём, и это важно для новичка по двум причинам.

Первая: начиная с этой версии Python по умолчанию читает и пишет текстовые файлы в кодировке UTF-8 на любой системе. Раньше на Windows русские буквы в файлах иногда превращались в "кракозябры", и новички тратили вечер на поиск причины. Теперь таких историй станет заметно меньше.

Вторая: Python 3.15 ещё понятнее объясняет ошибки. Если вы по привычке из другого языка напишете push вместо append, он прямо подскажет правильное слово. Подробнее об этом - в главах 1 и 5.

Если у вас установлена более старая версия, большая часть примеров всё равно заработает. Но некоторые новинки (например, тип frozendict из главы 6) появились только в 3.15. Поэтому лучше поставить свежую версию: в первой главе показано, как это сделать.

Код примеров

Короткие программы из первых глав лучше набирать руками. Звучит старомодно, зато так в голове остаётся в разы больше: пальцы запоминают, где ставятся скобки и двоеточия, а глаза учатся замечать опечатки. Ошибки при наборе тоже полезны: их разбор - лучшая тренировка.

Начиная с главы 13 понадобятся дополнительные библиотеки, а как их ставить, рассказано в главе 11.

Что делать, если не получается

Не получаться будет. Это не признак того, что программирование "не ваше", это обычный рабочий процесс. У опытных программистов программы тоже падают с ошибками по много раз в день, просто они быстрее находят причину. Вот порядок действий, который выручает почти всегда:

1. Прочитайте сообщение об ошибке до конца, особенно последнюю строку. Как это делать, мы разберём в главах 1 и 9.

2. Сравните свой код с листингом в книге символ за символом: скобки, кавычки, двоеточия, отступы.

3. Загляните в приложение В: там собраны частые ошибки новичков с объяснениями.

4. Поищите текст ошибки в интернете. Скорее всего, кто-то уже спрашивал о том же.

5. Спросите нейросеть, но с умом: как правильно спрашивать и проверять ответы, рассказано в главе 17.

6. Отложите задачу до завтра. Это не шутка: свежая голова находит ошибку за минуту там, где уставшая не нашла за час.

Как читать, чтобы был толк

Лучше понемногу, но регулярно: полчаса-час в день работают лучше, чем один длинный рывок в выходные. Читайте с открытым редактором кода. Запускайте каждый листинг, а потом ломайте его: поменяйте число, уберите скобку, допишите строку и посмотрите, что будет. Компьютер нельзя сломать неправильной программой из этой книги, а понимание приходит именно в таких экспериментах.

И ещё одно. Если какая-то тема не укладывается в голове с первого раза, это нормально. Прочитайте раздел ещё раз завтра, решите пару заданий, вернитесь к нему через главу. Многие вещи в программировании становятся понятными не тогда, когда о них прочитал, а тогда, когда они пригодились.

Барсик уже сидит на клавиатуре и смотрит выжидательно. Начнём с первой программы.

Глава 1. Первая программа

В этой главе вы установите Python, познакомитесь с редактором кода и напишете первую программу. А ещё научитесь главному навыку программиста, о котором редко говорят в начале: спокойно читать сообщения об ошибках. Ошибки будут. Хорошая новость в том, что Python почти всегда говорит, где именно споткнулся.

Что такое программа

Вспомните бабушкин рецепт блинов. "Взбить два яйца со стаканом молока, добавить щепотку соли и ложку сахара, всыпать муку, чтобы тесто было как жидкая сметана, жарить на горячей сковороде". Это последовательность действий, которая приводит к результату. Программа устроена так же: это список инструкций для компьютера.

Разница в исполнителе. Бабушка понимает, что такое "как жидкая сметана" и "по вкусу". Компьютер не понимает ничего, кроме того, что написано. Он не догадается, не додумает и не переспросит. Зато он выполнит инструкции быстро, без усталости и ровно так, как написано: хоть миллион раз подряд. Если в рецепте для компьютера сказано "жарить блин", а про "перевернуть" забыли, он честно сожжёт блин с одной стороны и будет считать, что всё сделал правильно.

Поэтому программирование - это умение объяснять. Объяснять так точно, чтобы понял исполнитель без капли здравого смысла. Звучит строго, но на деле это увлекательная игра: вы придумываете, как разложить задачу на простые шаги, а компьютер проверяет, нигде ли вы не схитрили.

Инструкции пишут на языке программирования (programming language). Это компромисс между человеческим языком и языком машины. Процессор понимает только числа, человек предпочитает слова, а язык программирования стоит посередине: он состоит из слов и знаков, но правила у него строгие, как у таблицы умножения.

Python - один из таких языков. Программу на Python выполняет интерпретатор (interpreter): особая программа, которая читает ваш код строку за строкой и делает то, что там написано. Когда говорят "установить Python", имеют в виду именно его.

Попробуйте сами 1.1. Запишите на бумаге алгоритм заваривания чая в пакетике для робота, который ничего не знает о чае. Распишите не меньше восьми шагов. Потом перечитайте их глазами робота: где он может понять вас неправильно? Например, что он сделает, если чайник уже полный?

Попробуйте сами 1.2. Придумайте три дела из своей жизни, которые могла бы взять на себя программа. Подсказка: ищите то, что повторяется, и то, что легко забыть.

Почему Python

Языков программирования сотни. Почему для первого знакомства стоит выбрать Python?

Во-первых, код на нём читается почти как текст. Сравните программу, которая выводит на экран "Привет, мир!", на языке Java:

```
public class Main {

public static void main(String[] args) {

System.out.println("Привет, мир!");

}

}
```

и на Python:

```
print("Привет, мир!")
```

Одна строка вместо пяти, и никаких загадочных слов, которые новичку приходится заучивать "пока просто так надо".

Во-вторых, Python умеет очень многое. На нём пишут сайты и Telegram-ботов, анализируют данные и строят графики, обучают нейросети, автоматизируют рутину в офисе, делают игры, управляют научными приборами. Он уже много лет входит в число самых популярных языков мира, поэтому на любой вопрос о нём в интернете найдётся ответ.

В-третьих, Python бесплатный и работает на Windows, macOS и Linux. К нему написаны сотни тысяч библиотек (library) - наборов готового кода. Кто-то уже решил за вас задачу отправки писем, рисования графиков или чтения таблиц Excel, и этим решением можно воспользоваться одной командой.

Честности ради, у Python есть и слабые стороны. Он медленнее языков вроде C++ в тяжёлых вычислениях, а мобильные приложения на нём почти не пишут. Но для первого языка и для огромного числа практических задач это лучший выбор. А когда придёт время учить второй язык, вы удивитесь, как много из Python в нём узнаётся.

Устанавливаем Python 3.15

Python не входит в Windows, его нужно поставить. Делается это один раз и занимает минут десять.

Установка на Windows

С недавних пор Python для Windows ставится через небольшую программу Python install manager (менеджер установки Python). Она устанавливает сам Python, следит за обновлениями и умеет держать на компьютере несколько версий сразу. Порядок действий такой.

1. Откройте в браузере страницу python.org/downloads/windows. В самом верху есть ссылка "Latest Python install manager" (последний менеджер установки). Перейдите по ней и скачайте файл "Installer (MSIX)". Другой путь: найти в Microsoft Store приложение "Python install manager" и нажать "Получить". Оба варианта одинаковые.

2. Запустите скачанный файл и нажмите "Install" (установить).

3. Откройте терминал. Щёлкните правой кнопкой мыши по значку "Пуск" и выберите "Терминал" (в Windows 10 это может быть "Windows PowerShell"). Откроется окно, в котором можно печатать команды.

4. Наберите команду и нажмите Enter:

py install 3.15

Менеджер скачает и установит Python 3.15. Если при этом он задаст вопросы о настройке компьютера (например, предложит добавить папку с командами в переменную PATH или разрешить длинные пути к файлам), соглашайтесь: введите y и нажмите Enter.

5. Закройте терминал, откройте его снова и проверьте, что всё получилось:

python --version

Python 3.15.1

Если вы видите номер версии, начинающийся с 3.15, Python установлен. Последняя цифра у вас может быть больше: это номер исправленного выпуска, они выходят раз в пару месяцев. Список установленных версий показывает команда py list.

Частая ошибка. Вместо версии Python открывается Microsoft Store или терминал отвечает, что имя "python" не распознано (в старой командной строке cmd: "python не является внутренней или внешней командой"). Сначала закройте и снова откройте терминал: новые команды видны только в новом окне. Если не помогло, нажмите "Пуск", найдите "Управление псевдонимами выполнения приложений" и проверьте, что переключатели у пунктов "Python (default)" и "Python install manager" включены. Если они уже включены, выключите и включите их снова.

Совет. На странице загрузок для Python 3.15 ещё лежит и старый установщик "Windows installer (64-bit)". Им тоже можно пользоваться, но в первом окне обязательно поставьте галочку "Add python.exe to PATH". Без неё команда python в терминале работать не будет. Для версии 3.16 его уже не выпустят, поэтому в книге мы используем менеджер установки.

Если у вас macOS или Linux. На macOS скачайте установщик для macOS со страницы python.org/downloads/macos (файл .pkg) и пройдите обычную установку. Проверка - в программе "Терминал": python3 --version. На Linux Python обычно уже есть, но часто старой версии. Проще всего поставить 3.15 через утилиту uv: о ней рассказано в главе 11, а сама установка - две команды: curl -LsSf https://astral.sh/uv/install.sh | sh и uv python install 3.15. Такой Python запускается командой python3.15. Везде, где в книге написано python, на macOS пишите python3, а на Linux - python3.15.

Интерактивный режим

Python установлен, давайте с ним поговорим. Наберите в терминале одно слово и нажмите Enter:

python

Появится несколько строк о версии и приглашение из трёх угловых скобок >>>. Это интерактивный режим (interactive mode), его ещё называют оболочкой или REPL (от английского read-eval-print loop: "прочитать, выполнить, напечатать, повторить"). Python ждёт команду, выполняет её и сразу показывает результат. Попробуйте:

```
>>> 2 + 2

4

>>> 7 * 8

56

>>> 100 / 3

33.333333333333336

>>> 2 ** 10

1024

>>> (1500 + 230) * 2

3460
```

Перед вами калькулятор, причём неплохой. Знак * означает умножение, / - деление, а ** - возведение в степень: 2 ** 10 - это два в десятой степени. Скобки работают как в школе.

Теперь текст. Текст в Python пишут в кавычках:

```
>>> "Привет"

'Привет'

>>> print("Привет")

Привет

>>> "Барсик" + " и сосиска"

'Барсик и сосиска'

>>> "Мяу" * 3

'МяуМяуМяу'
```

Заметили разницу между первой и второй строкой? Если просто написать текст, оболочка покажет его в кавычках: так она подчёркивает, что это именно текст, а не число и не

команда. А print выводит сам текст, без кавычек, как его увидит человек. Знак + склеивает два куска текста, а умножение на число повторяет текст нужное количество раз. Барсик одобряет: так мяукать можно бесконечно.

Чтобы выйти из интерактивного режима, наберите exit и нажмите Enter. Можно и сочетанием клавиш: на Windows Ctrl+Z, затем Enter; на macOS и Linux Ctrl+D.

Интерактивный режим хорош для быстрых проб: проверить, как работает команда, посчитать что-нибудь, вспомнить забытое. Но всё, что вы в нём набрали, исчезает после выхода. Для настоящих программ нужен файл и редактор кода.

Попробуйте сами 1.3. Посчитайте в интерактивном режиме, сколько минут в неделе и сколько секунд в году (возьмите год в 365 дней).

Попробуйте сами 1.4. Что выведут команды "Барсик" + "!" и "Мур" * 5? Подумайте, прежде чем нажать Enter. А потом попробуйте "Мур" + 5 и прочитайте, что ответит Python. Что, по-вашему, ему не понравилось?

Редактор кода

Программы на Python хранятся в обычных текстовых файлах с расширением .py. В принципе, писать их можно хоть в Блокноте. Но удобнее в редакторе кода: он подсвечивает команды разными цветами, сам ставит отступы, подчёркивает опечатки и запускает программу одной кнопкой.

Visual Studio Code

В книге мы будем работать в Visual Studio Code (коротко VS Code). Это бесплатный редактор, который любят и новички, и профессионалы.

1. Скачайте VS Code с сайта code.visualstudio.com и установите. На шаге "Выберите дополнительные задачи" поставьте галочку "Добавить в PATH" и галочки у действий "Открыть с помощью Code": они пригодятся.

2. Запустите VS Code. Слева есть вертикальная панель со значками. Нажмите на значок из четырёх квадратиков (это "Расширения", или сочетание Ctrl+Shift+X), найдите в поиске Python и установите расширение Python от Microsoft. Оно научит редактор понимать наш язык.

3. Если хочется меню на русском, там же найдите и установите "Russian Language Pack for Visual Studio Code", а потом перезапустите редактор.

Теперь заведём папку для программ. Создайте папку C:\pyprojects: откройте Проводник, зайдите на диск C:, нажмите правой кнопкой мыши на пустое место и выберите "Создать → Папку". В VS Code выберите в меню "Файл → Открыть папку" и откройте её. Если редактор спросит, доверяете ли вы авторам файлов в этой папке, ответьте "Да": автор здесь вы.

Совет. Включите в Проводнике показ расширений файлов: "Вид → Показать → Расширения имён файлов" (в Windows 10: вкладка "Вид", галочка "Расширения имён файлов"). Иначе легко получить файл hello.py.txt, который выглядит как hello.py, но Python его не запустит.

IDLE

Если ставить VS Code пока не хочется, есть запасной вариант. Вместе с Python устанавливается простенький редактор IDLE. Найдите его в меню "Пуск" по слову IDLE. Откроется окно интерактивного режима, а через меню "File → New File" можно создать файл программы. Запускается программа клавишей F5, результат появляется в первом окне. Для первых глав IDLE вполне хватит, а к проектам всё равно лучше перейти на VS Code.

Если у вас macOS или Linux. VS Code есть и для этих систем, ставится с того же сайта. Папку для программ удобно создать в домашней папке: ~/pyprojects.

Первая программа

По традиции, которой уже полвека, первая программа на новом языке выводит на экран "Привет, мир!". В VS Code нажмите "Файл → Создать файл", назовите его hello.py и наберите одну строку.

Листинг 1.1. hello.py

```
print("Привет, мир!")
```

Сохраните файл (Ctrl+S). Запустить программу можно двумя способами. Первый: в VS Code в правом верхнем углу есть кнопка с треугольником "Run Python File" (запустить файл Python). Второй: открыть терминал (в VS Code это меню "Терминал → Создать терминал") и набрать команду python с именем файла:

```
python hello.py
```

Если терминал открыт не в папке с программой, сначала перейдите в неё командой cd C:\pyprojects. Каким бы способом вы ни запустили программу, в терминале появится:

```
Привет, мир!
```

Поздравляем, вы программист! Без шуток: вы написали инструкцию, и компьютер её выполнил.

Разберём, что тут написано. print - это функция (function), то есть готовая команда, которая что-то делает. Функция print выводит на экран то, что ей передали в скобках. Скобки после имени функции обязательны: именно они говорят Python "выполни". Внутри скобок - строка (string), то есть текст в кавычках. Кавычки не выводятся, они только показывают, где текст начинается и где заканчивается.

Строку можно заключать и в двойные кавычки, и в одинарные: print('Привет, мир!') работает точно так же. В книге мы будем писать двойные. Главное, чтобы открывающая и закрывающая кавычки были одинаковыми.

Несколько строк

Программа может состоять из многих строк. Python выполняет их по порядку, сверху вниз.

Листинг 1.2. about_cat.py

```
print("Знакомьтесь: Барсик.")

print("Возраст:", 4, "года")

print("Любимая еда: сосиски, которые плохо лежат.")

print()

print("Сколько сосисок Барсик съел за неделю:", 2 * 7)
```

```
Знакомьтесь: Барсик.

Возраст: 4 года

Любимая еда: сосиски, которые плохо лежат.


Сколько сосисок Барсик съел за неделю: 14
```

Здесь три новых приёма. Во второй строке функции print передано несколько значений через запятую: текст, число и снова текст. Она выводит их подряд и ставит между ними пробел. В четвёртой строке print() без всего внутри выводит пустую строку: так удобно разделять части вывода. А в последней строке Python сначала посчитал 2 * 7 и только потом вывел результат. Кавычек вокруг 2 * 7 нет, поэтому это не текст, а вычисление.

Сравните: print("2 * 7") выведет 2 * 7, а print(2 * 7) выведет 14. Кавычки решают всё.

Картинка из символов

Раз print выводит любые символы, ими можно рисовать. Скоро Новый год, нарисуем ёлку.

Листинг 1.3. yolka.py

```
print("  *")

print(" ***")

print("*****")

print("*******")

print("*********")

print("  | |")

print()

print("С наступающим!")
```

```
  *

 ***

*****

*******

*********

  | |


С наступающим!
```

Пробелы в начале строк внутри кавычек - часть текста, поэтому ёлка ровная. Обратите внимание: сами строки кода начинаются с print без отступа. Отступы в начале строк кода в Python имеют особый смысл, и мы ещё до него дойдём.

Кавычки внутри текста

А как вывести текст, в котором сами есть кавычки? Например, Барсик сказал: "Мяу!". Если написать print("Барсик сказал: "Мяу!""), Python решит, что строка закончилась перед

словом Мяу, и запутается. Выход простой: снаружи взять одинарные кавычки, а внутри оставить двойные.

Листинг 1.4. quotes.py

```
print('Барсик сказал: "Мяу!"')

print("Хозяйка ответила: 'Сосисок больше нет.'")
```

```
Барсик сказал: "Мяу!"

Хозяйка ответила: 'Сосисок больше нет.'
```

Внешние кавычки отмечают границы строки, внутренние стали обычными символами текста.

Попробуйте сами 1.5. Напишите программу about_me.py, которая выводит на трёх строках ваше имя, город и любимую еду.

Попробуйте сами 1.6. Нарисуйте символами домик, кота или ракету. Чем смешнее, тем лучше.

Попробуйте сами 1.7. Выведите одной программой строки В сутках 24 часа и В неделе 168 часов, но число 168 пусть посчитает Python: напишите вычисление, а не готовый ответ.

Попробуйте сами 1.8. Выведите фразу Мама сказала: "Не корми кота со стола!" вместе с кавычками.

Как читать сообщения об ошибках

Теперь самое полезное. Вы обязательно будете ошибаться: пропускать скобки, путать кавычки, опечатываться в словах. Все так делают, включая авторов книг по программированию. Разница между новичком и опытным программистом не в том, что у второго нет ошибок, а в том, что он умеет быстро их читать.

Давайте ошибёмся нарочно.

Листинг 1.5. error_name.py

```
print("Начинаем")

pirnt("Привет, мир!")

print("Заканчиваем")
```

Во второй строке опечатка: pirnt вместо print. Запустим:

```
Начинаем

Traceback (most recent call last):

File "C:\pyprojects\error_name.py", line 2, in <module>

pirnt("Привет, мир!")

^^^^^

NameError: name 'pirnt' is not defined. Did you mean: 'print'?
```

Не пугайтесь английских слов, разберём сообщение по частям. Такое сообщение называется трассировкой (traceback): это отчёт о том, где программа была, когда случилась беда.

Traceback (most recent call last): - заголовок: "Трассировка (самый последний вызов внизу)". Пока его можно пропускать.

File "C:\pyprojects\error_name.py", line 2, in <module> - где ошибка: файл и номер строки, здесь вторая. Слова in <module> значат "в основном тексте программы", а не внутри какой-то функции.

Дальше Python показывает саму строку с ошибкой и подчёркивает значками ^ место, которое ему не понравилось.

Последняя строка - самое главное: тип ошибки и объяснение. NameError - ошибка имени. name 'pirnt' is not defined - "имя pirnt не определено", то есть Python не знает такого слова. А дальше подсказка: Did you mean: 'print'? - "Возможно, вы имели в виду print?".

Ещё одна важная деталь: строка "Начинаем" успела вывестись. Python выполнил первую строку, споткнулся на второй и остановился. До третьей строки дело не дошло.

Правило чтения трассировки: начинайте с последней строки. Она говорит, что случилось. Потом поднимайтесь выше и смотрите, где это случилось.

Синтаксические ошибки

Бывают ошибки другого рода, когда нарушены сами правила записи. Например, забыли закрыть скобку.

Листинг 1.6. error_bracket.py

```
print("Начинаем")
```

```
print("Привет, мир!"

print("Заканчиваем")
File "C:\pyprojects\error_bracket.py", line 2

print("Привет, мир!"

^

SyntaxError: '(' was never closed
```

SyntaxError - синтаксическая ошибка, нарушение правил языка. Объяснение: '(' was never closed - "скобка '(' так и не была закрыта". Значок ^ указывает на ту самую незакрытую скобку.

Заметьте разницу с прошлым примером: здесь не вывелось даже "Начинаем". Перед запуском Python читает весь файл целиком и проверяет, соблюдены ли правила записи. Если нет, программа не запускается вовсе. Это как с сочинением: учительница не станет вникать в смысл, пока в нём половина слов без окончаний.

Похожая история с кавычками.

Листинг 1.7. error_quote.py

```
print("Привет, мир!)
File "C:\pyprojects\error_quote.py", line 1

print("Привет, мир!)

^

SyntaxError: unterminated string literal (detected at line 1)
```

unterminated string literal - "незавершённая строка": Python дошёл до конца строки кода, а закрывающую кавычку так и не встретил. В скобках уточнение: обнаружено в строке 1.

И третья классика - лишний пробел в начале строки.

Листинг 1.8. error_indent.py

```
print("Начинаем")

print("Привет, мир!")
File "C:\pyprojects\error_indent.py", line 2

print("Привет, мир!")

IndentationError: unexpected indent
```

IndentationError: unexpected indent - "ошибка отступа: неожиданный отступ". В Python пробелы в начале строки - это не украшение, а часть смысла программы. Пока все наши строки должны начинаться с самого края. Зачем нужны отступы, вы узнаете в главе 3.

Ещё два частых промаха

Забыть кавычки вокруг текста:

Листинг 1.9. error_no_quotes.py

```
print(Привет)
Traceback (most recent call last):
```

File "C:\pyprojects\error_no_quotes.py", line 1, in <module>

print(Привет)

^^^^^^

NameError: name 'Привет' is not defined

Без кавычек Python считает Привет не текстом, а именем чего-то (позже мы узнаем, что так называют переменные), и честно сообщает, что такого имени не знает.

А если вы раньше видели старые учебники, рука может написать print без скобок:

Листинг 1.10. error_old_print.py

print "Привет"

File "C:\pyprojects\error_old_print.py", line 1

print "Привет"

^^^^^^^^^^^^^^

SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?

Так писали в Python 2, который давно не поддерживается. Python узнаёт эту привычку и прямо подсказывает: Did you mean print(...)? - "Возможно, вы имели в виду print(...)?".

Новое в Python 3.15. Python давно умеет подсказывать "Did you mean", а в версии 3.15 подсказок стало ещё больше. Например, если по привычке из другого языка написать название команды из JavaScript или Java, Python назовёт её питоновский аналог. Такие подсказки мы увидим в главах 5 и 7.

Маленький словарик ошибок

Английский в сообщениях об ошибках довольно однообразный. Вот слова, которые встречаются чаще всего:

Error - ошибка; Syntax - синтаксис, правила записи;
not defined - не определено, неизвестно;
was never closed - так и не закрыто;
unterminated - незавершённый;
unexpected - неожиданный;
invalid - недопустимый, неправильный;
expected - ожидалось;
Did you mean - возможно, вы имели в виду.

Если слово незнакомо, смело копируйте всю последнюю строку трассировки в переводчик или в поиск. Сообщения стандартные, и по каждому из них в интернете есть объяснения.

Попробуйте сами 1.9. Сломайте программу hello.py тремя способами: уберите закрывающую кавычку, уберите закрывающую скобку, напишите Print с большой буквы. Каждый раз запускайте и читайте сообщение. Какие из трёх ошибок синтаксические?

Попробуйте сами 1.10. В этой программе три ошибки. Найдите их, не запуская программу, а потом запустите и проверьте себя по сообщениям Python.

print("Список покупок:")

print("молоко)

```
prnt("хлеб")

print("корм для Барсика")
```

Итоги

Программа - это точная последовательность инструкций. Python выполняет её строку за строкой, сверху вниз; делает это интерпретатор.

Python 3.15 на Windows ставится через Python install manager; проверка установки - команда python --version в терминале.

Интерактивный режим (>>>) подходит для быстрых проб, а программы пишут в файлах .py в редакторе кода: VS Code или IDLE.

Функция print() выводит на экран текст и числа. Текст пишут в кавычках, несколько значений передают через запятую.

При синтаксической ошибке программа не запускается вовсе, при остальных ошибках выполняется до строки с ошибкой.

Трассировку читают снизу вверх: последняя строка говорит, что случилось, строки выше - где именно. Подсказки "Did you mean" часто сразу называют исправление.

Глава 2. Переменные и типы данных

В первой главе программа просто выводила готовый текст. Это как открытка, где всё напечатано заранее. Настоящая программа работает с данными: запоминает имя, считает деньги, спрашивает возраст и что-то с ним делает. В этой главе вы научитесь хранить данные в переменных, считать, работать с текстом и спрашивать человека за клавиатурой.

Переменные

Представьте полку в кладовке на даче. На ней банки с вареньем, и на каждой ярлычок: "Малина", "Крыжовник", "Слива 2025". Ярлычок помогает найти нужную банку, не открывая все подряд. Переменная (variable) в Python - это такой ярлычок: имя, которое прикреплено к значению.

Листинг 2.1. variables.py

```
cat_name = "Барсик"

cat_age = 4

print(cat_name, "живёт с нами уже", cat_age, "года")


cat_age = 5

print("А в марте ему исполнится", cat_age)
```

Барсик живёт с нами уже 4 года

А в марте ему исполнится 5

Строка cat_name = "Барсик" читается так: "прикрепить имя cat_name к строке Барсик". Знак = в Python означает не "равно", а присваивание (assignment): справа значение, слева имя, которое к нему прикрепляют. Потом по имени можно получить значение: print(cat_name) выведет Барсик.

Значение переменной можно заменить. В пятой строке ярлычок cat_age перевесили с числа 4 на число 5. Старое значение забыто, и второй print выводит уже 5.

Раз = - это присваивание, а не равенство, в Python совершенно законна строка, от которой учительница математики упала бы в обморок:

```
>>> sausages = 3

>>> sausages = sausages + 1

>>> sausages

4
```

Python сначала вычисляет правую часть: берёт текущее значение sausages (3), прибавляет 1 и получает 4. Потом прикрепляет имя sausages к результату. Барсик стянул ещё одну сосиску, и переменная это учла.

Обратите внимание на последнюю строку: в интерактивном режиме достаточно написать имя переменной, и Python покажет её значение. В программе из файла так не получится, там нужен print.

Как называть переменные

Правила для имён простые:

имя состоит из букв, цифр и знака подчёркивания _;

имя не может начинаться с цифры;

пробелы внутри имени нельзя;

большие и маленькие буквы различаются: cat, Cat и CAT - три разные переменные;

нельзя занимать служебные слова (keywords) Python, такие как if, for, class, True. Всего их 35, и редактор подсвечивает их особым цветом, так что перепутать трудно.

Вот что бывает, если правила нарушить:

```
>>> 2cats = 2

File "<python-input-0>", line 1

2cats = 2

^

SyntaxError: invalid decimal literal

>>> my cat = "Барсик"

File "<python-input-1>", line 1

my cat = "Барсик"

^^^

SyntaxError: invalid syntax

>>> class = "5Б"

File "<python-input-2>", line 1

class = "5Б"

^

SyntaxError: invalid syntax
```

Кроме правил есть и договорённости, которых придерживаются все программисты на Python. Имена переменных пишут маленькими английскими буквами, а слова разделяют подчёркиванием: cat_name, price_per_kg, total_sum. Такой стиль называют snake_case, "змеиный регистр": слова будто ползут друг за другом по земле.

Python разрешает и русские имена: строка кот = "Барсик" сработает. Но так не принято. Английские имена понимают программисты всего мира, и не нужно сто раз в минуту переключать раскладку. Русский язык оставим для текста, который видит человек: подсказок, сообщений и комментариев.

И главное правило: имя должно объяснять, что лежит внутри. Сравните a = 45 и tomato_price = 45. Через неделю вы сами не вспомните, что такое a, а смысл tomato_price ясен сразу. Длинное понятное имя лучше короткого загадочного.

Попробуйте сами 2.1. Какие из этих имён Python примет, а какие нет: price, 2price, price2, total price, total_price, for, _secret, Барсик? Проверьте себя в интерактивном режиме.

Попробуйте сами 2.2. Заведите переменные breakfast, lunch и dinner с названиями блюд и выведите меню на день одной функцией print.

Числа

В Python два основных вида чисел. Целые числа (integer, коротко int) пишутся как обычно: 4, -15, 1000. Дробные числа (floating point, коротко float, "числа с плавающей точкой") пишутся через точку, а не через запятую: 3.5, 0.25, -12.0. Узнать тип значения помогает функция type:

```
>>> type(4)

<class 'int'>

>>> type(4.5)

<class 'float'>

>>> type("Барсик")

<class 'str'>
```

int - целое число, float - дробное, str (от string) - строка. Тип (type) значения определяет, что с ним можно делать: числа складывают и умножают, а у строк свои умения.

Арифметика

С числами работают знакомые действия:

Действие
Знак
Пример
Результат

сложение
+
7 + 2
9

вычитание
-
7 - 2
5

умножение
*
7 * 2
14

деление
/
7 / 2
3.5

целая часть от деления
//
7 // 2
3

остаток от деления
%
7 % 2
1

возведение в степень
**
7 ** 2
49

Порядок действий как в школе: сначала степень, потом умножение и деление, потом сложение и вычитание. Скобки меняют порядок. Посчитаем покупку семян для дачи.

Листинг 2.2. seeds.py

```
tomato_price = 45

cucumber_price = 38.5

tomato_packs = 3

cucumber_packs = 2


total = tomato_price * tomato_packs + cucumber_price * cucumber_packs

print("Итого:", total, "руб.")

print("Сдача с 1000 рублей:", 1000 - total, "руб.")
```

Итого: 212.0 руб.

Сдача с 1000 рублей: 788.0 руб.

Семена огурцов стоят 38.5 рубля, это дробное число. Как только в вычислении участвует хоть одно дробное число, результат тоже становится дробным. Поэтому итог 212.0, а не 212.

Есть и ещё одно правило: обычное деление / всегда даёт дробное число, даже если делится нацело.

```
>>> 10 / 2

5.0

>>> 10 // 2

5

>>> 2 ** 100
```

```
126765060022822940149670320537 6

>>> 1_000_000 * 3

3000000
```

2 ** 100 - число из тридцати одной цифры, и Python посчитал его точно. Целые числа в Python могут быть любой длины, пока хватает памяти компьютера. А подчёркивания внутри числа 1_000_000 нужны только для удобства чтения, как пробелы между разрядами в бухгалтерии. Python их просто пропускает.

Целое деление и остаток

Знаки // и % кажутся странными, пока не понадобятся. А нужны они постоянно. Пусть у нас 17 сосисок и 5 котов, которых надо накормить поровну.

Листинг 2.3. sausages.py

```
sausages = 17

cats = 5


per_cat = sausages // cats

left = sausages % cats


print("Каждому коту:", per_cat)

print("Осталось:", left)

print("Остаток, по традиции, достаётся Барсику.")
```

```
Каждому коту: 3

Осталось: 2

Остаток, по традиции, достаётся Барсику.
```

17 // 5 - сколько раз 5 целиком помещается в 17, то есть 3. 17 % 5 - сколько остаётся, то есть 2. Остаток от деления помогает узнать, чётное ли число (n % 2 равно 0 у чётных), перевести минуты в часы и минуты, разложить предметы по коробкам. Мы ещё не раз его встретим.

Сокращённые записи

Строки вида total = total + 100 встречаются так часто, что для них придумали сокращение: total += 100. Так же работают -=, *= и /=.

```
>>> money = 1000

>>> money -= 212

>>> money
```

```
788

>>> money *= 2

>>> money

1576
```

Сюрприз с дробными числами

А теперь фокус, который удивляет почти всех:

```
>>> 0.1 + 0.2

0.30000000000000004

>>> round(0.1 + 0.2, 2)

0.3

>>> round(7.456, 1)

7.5
```

Это не ошибка Python. Компьютер хранит дробные числа в двоичном виде, и некоторые простые для нас дроби, например 0.1, в двоичной записи бесконечны, как 1/3 в десятичной (0.3333...). Поэтому в последнем знаке появляется крошечная погрешность. Для бытовых расчётов она не страшна: результат округляют функцией round, которой передают число и сколько знаков оставить после точки. Для точных денежных расчётов в Python есть особый тип Decimal, о нём - в главе 11.

Попробуйте сами 2.3. Стену дачного домика площадью 37 квадратных метров надо покрасить в два слоя. Одной банки краски хватает на 9 квадратных метров одного слоя. Сколько целых банок понадобится? Посчитайте программой. Подсказка: если краски не хватает чуть-чуть, банку всё равно придётся купить целую.

Попробуйте сами 2.4. Фильм идёт 135 минут. Выведите его длительность в виде "2 ч 15 мин", используя // и %.

Попробуйте сами 2.5. Посчитайте в интерактивном режиме 0.1 + 0.1 + 0.1 и 0.1 * 3. Совпали ли результаты с ожиданием? Округлите их до двух знаков.

Строки

С текстом мы уже знакомы: строка (string) - это символы в кавычках. Строки можно склеивать знаком + и повторять знаком *, а функция len показывает длину строки, то есть количество символов в ней:

```
>>> greeting = "Привет, " + "Барсик"

>>> greeting

'Привет, Барсик'

>>> len(greeting)

14

>>> "-" * 20

'--------------------'
```

В длину входят все символы, включая пробелы и знаки препинания: "Привет, Барсик" - это 14 символов.

Особые символы

Как вывести текст в две строки одной командой print? В строку можно вставить особые символы, которые начинаются с обратной косой черты \. Самые нужные:

\n - перевод строки (n от new line, "новая строка");
\t - табуляция, большой отступ;
\" и \' - кавычка, которая не закрывает строку;
\\ - сама обратная косая черта.

Листинг 2.4. special_chars.py

```
print("Список покупок:\n- молоко\n- хлеб\n- корм для кота")

print("Товар\tЦена")

print("Сыр\t450")

print("Кот сказал: \"Мяу!\"")

print("Файл лежит в папке C:\\pyprojects")
```

```
Список покупок:

- молоко

- хлеб

- корм для кота

Товар Цена
```

```
Сыр 450

Кот сказал: "Мяу!"

Файл лежит в папке C:\pyprojects
```

Частая ошибка. Пути к файлам в Windows пишутся с обратной косой чертой, и это ловушка. В строке "C:\new\table.txt" Python увидит \n (перевод строки) и \t (табуляцию), и путь развалится. Выхода три: удваивать черту ("C:\\new\\table.txt"), писать прямую черту ("C:/new/table.txt", Python на Windows её понимает) или ставить перед строкой букву r: r"C:\new\table.txt". Буква r (от raw, "сырой") велит Python не трогать обратные черты. В главе 10 мы узнаем ещё более удобный способ работать с путями.

Многострочный текст

Если текст длинный и в нём много строк, удобнее заключить его в тройные кавычки. Тогда переводы строк сохраняются как есть:

Листинг 2.5. postcard.py

```
postcard = """Дорогая бабушка!

Поздравляю с днём рождения!

Желаю здоровья, хорошего урожая

и поменьше кротов на грядках.

Твой внук Дима и кот Барсик"""


print(postcard)
Дорогая бабушка!

Поздравляю с днём рождения!

Желаю здоровья, хорошего урожая

и поменьше кротов на грядках.

Твой внук Дима и кот Барсик
```

f-строки

Чаще всего строку собирают из кусочков: текста и значений переменных. Можно склеивать плюсом, но для этого числа придётся превращать в текст функцией str, а пробелы расставлять вручную. Получается громоздко: "Коту " + str(cat_age) + " года". Гораздо удобнее f-строки (f-strings). Перед открывающей кавычкой ставят букву f, а значения подставляют в фигурных скобках прямо внутри текста:

Листинг 2.6. fstrings.py

```
cat_name = "Барсик"
```

```
cat_age = 4

weight = 5.678


print(f"Кота зовут {cat_name}, ему {cat_age} года.")

print(f"Через год ему будет {cat_age + 1}.")

print(f"Вес: {weight} кг, после округления: {weight:.1f} кг.")

print(f"Стоимость корма на месяц: {30 * 65.5:.2f} руб.")
```

```
Кота зовут Барсик, ему 4 года.

Через год ему будет 5.

Вес: 5.678 кг, после округления: 5.7 кг.

Стоимость корма на месяц: 1965.00 руб.
```

В фигурных скобках может быть не только имя переменной, но и вычисление: {cat_age + 1}. А после двоеточия можно указать формат. Запись :.1f значит "показать дробное число с одним знаком после точки", :.2f - с двумя. Для денег формат :.2f нужен почти всегда: "1965.00 руб." выглядит привычнее, чем "1965.0 руб.". Других форматов много, мы разберём их в главе 7.

Ещё один приём пригодится при поиске ошибок. Если после имени в фигурных скобках поставить знак =, Python выведет и имя, и значение:

```
>>> cat_age = 4

>>> print(f"{cat_age=}")

cat_age=4
```

Попробуйте сами 2.6. Заведите переменные с названием товара, ценой за килограмм и весом покупки. Выведите f-строкой фразу вида "Помидоры: 1.5 кг по 180 руб. = 270.00 руб.".

Попробуйте сами 2.7. Одной командой print выведите табличку из трёх строк "Имя - Возраст" для трёх членов семьи, используя \n и \t.

Попробуйте сами 2.8. Выведите f-строкой длину своего полного имени: "В имени Мария 5 букв". Используйте len прямо внутри фигурных скобок.

Логический тип

Есть тип совсем простой: у него всего два значения, True (истина) и False (ложь). Он называется логическим (boolean, коротко bool, в честь математика Джорджа Буля). Логические значения получаются, когда мы что-то сравниваем:

```
>>> 5 > 3

True

>>> 2 + 2 == 5

False

>>> "Барсик" == "барсик"

False

>>> is_hungry = True

>>> type(is_hungry)

<class 'bool'>
```

Двойное равенство == - это сравнение: "равны ли значения?". Не путайте его с одинарным =, которое присваивает. "Барсик" с большой буквы и "барсик" с маленькой для Python разные строки, поэтому сравнение даёт False.

Логический тип станет главным героем следующей главы, где программа начнёт принимать решения: если кот голодный, покормить, иначе не кормить.

Попробуйте сами 2.9. Не запуская Python, предскажите результат: 10 / 2 == 5, 7 // 2 == 3.5, "кот" * 2 == "коткот", len("мяу") == 3. Проверьте себя в интерактивном режиме.

Ввод с клавиатуры

Пора научить программу слушать. Функция input (ввод) выводит подсказку и ждёт, пока человек напечатает ответ и нажмёт Enter. Напечатанное она возвращает как строку, и его можно сохранить в переменной.

Листинг 2.7. greeting.py

```
name = input("Как вас зовут? ")

city = input("Из какого вы города? ")

print(f"Привет, {name}! Как погода в городе {city}?")
```

```
Как вас зовут? Наташа

Из какого вы города? Тверь

Привет, Наташа! Как погода в городе Тверь?
```

Заметьте пробел в конце подсказки "Как вас зовут? ". Без него ответ прилипнет к вопросительному знаку: Как вас зовут?Наташа.

Ввод - всегда строка

Попробуем посчитать, сколько человеку будет лет через год:

Листинг 2.8. next_year_bad.py

```
age = input("Сколько вам лет? ")

print(f"Через год вам будет {age + 1}")
```

```
Сколько вам лет? 25

Traceback (most recent call last):

File "C:\pyprojects\next_year_bad.py", line 2, in <module>

print(f"Через год вам будет {age + 1}")

~~~~^~~

TypeError: can only concatenate str (not "int") to str
```

Ошибка TypeError: ошибка типа. "Можно присоединять к строке только строку (а не int)". Функция input всегда возвращает строку, даже если человек ввёл цифры. Для Python "25" - это два символа, двойка и пятёрка, а не число двадцать пять. Прибавить к ним единицу он не может.

Чтобы превратить строку в число, нужна функция int (для целых) или float (для дробных):

Листинг 2.9. next_year.py

```
age = int(input("Сколько вам лет? "))

print(f"Через год вам будет {age + 1}")
```

```
Сколько вам лет? 25
```

Через год вам будет 26

Здесь функции вложены друг в друга, как матрёшки. Python сначала выполняет внутреннюю: input спрашивает возраст и возвращает строку "25". Потом внешнюю: int превращает строку "25" в число 25. Число и сохраняется в переменной age.

Обратное превращение тоже бывает нужно: функция str превращает число в строку. Такие превращения называют преобразованием типов (type conversion).

```
>>> int("42") + 1

43

>>> float("3.5") * 2

7.0

>>> str(42) + "1"

'421'

>>> float("3,5")

Traceback (most recent call last):

File "<python-input-3>", line 1, in <module>

float("3,5")

~~~~~^^^^^^^

ValueError: could not convert string to float: '3,5'
```

Последний пример - ловушка для всех, кто привык к русской записи чисел. В Python дробную часть отделяют точкой, и строку "3,5" функция float не понимает. Если программа просит ввести дробное число, подскажите человеку в тексте вопроса: "Введите вес через точку".

Что будет, если на вопрос о возрасте ответить словами?

```
Сколько вам лет? двадцать пять

Traceback (most recent call last):

File "C:\pyprojects\next_year.py", line 1, in <module>

age = int(input("Сколько вам лет? "))

~~~^^^^^^^^^^^^^^^^^^^^^^^^^^

ValueError: invalid literal for int() with base 10: 'двадцать пять'
```

ValueError - ошибка значения: "недопустимая запись для int() по основанию 10". Строку "двадцать пять" нельзя превратить в число. Пока примем это как есть и будем вводить числа аккуратно. В главе 9 мы научим программу вежливо переспрашивать, а не падать.

Кошачьи годы

Соберём всё вместе. Ветеринары считают, что первый год жизни кошки соответствует примерно 15 человеческим годам, второй добавляет ещё 9, а каждый следующий - по 4. Значит, двухлетний кот по-человечески ровесник 24-летнего, а дальше прибавляем по 4 года за каждый кошачий.

Листинг 2.10. cat_years.py

```
# Пересчёт возраста кота в человеческие годы

# (формула для взрослых котов, от двух лет)

name = input("Как зовут кота? ")

cat_age = int(input("Сколько ему лет? "))


human_age = 24 + (cat_age - 2) * 4

print(f"По-человечески {name} примерно {human_age}-летний.")
```

```
Как зовут кота? Барсик

Сколько ему лет? 4

По-человечески Барсик примерно 32-летний.
```

Четырёхлетний Барсик - ровесник 32-летнего человека. Это многое объясняет: он уже понял, где хранятся сосиски, но ещё не утратил азарта их добывать. Для котят младше двух лет формула не подходит. Как научить программу выбирать формулу в зависимости от возраста, вы узнаете в следующей главе.

Попробуйте сами 2.10. Напишите программу, которая спрашивает длину и ширину грядки в метрах (дробные числа) и выводит её площадь с двумя знаками после точки.

Попробуйте сами 2.11. Напишите программу-конвертер: она спрашивает сумму в рублях и курс доллара, а выводит, сколько долларов можно купить. Результат округлите до центов, то есть до двух знаков.

Попробуйте сами 2.12. Измените программу next_year.py так, чтобы она спрашивала ещё и имя, а отвечала фразой "Маша, через 10 лет вам будет 35".

Комментарии

В листинге 2.10 первые две строки начинаются со знака #. Это комментарии (comments): всё, что стоит после # до конца строки, Python пропускает. Комментарии пишут для людей: для коллег и для себя через полгода.

Комментарий может стоять и в конце строки кода:

Листинг 2.11. comments.py

```
price = 180 # цена помидоров за килограмм, руб.

weight = 1.5 # сколько купили, кг


# Скидка 10% по карте магазина

total = price * weight * 0.9

print(f"К оплате: {total:.2f} руб.")

# print("Эта строка не выполнится: она закомментирована")
```

```
К оплате: 243.00 руб.
```

Последняя строка программы закомментирована: перед ней стоит #, и Python её не выполняет. Так временно отключают кусок кода, не удаляя его. В VS Code это делает сочетание клавиш Ctrl+/ - оно ставит или убирает # у всех выделенных строк.

Хороший комментарий объясняет, зачем что-то сделано, а не пересказывает код. Комментарий # умножаем на 0.9 бесполезен: это и так видно. А # Скидка 10% по карте магазина объясняет, откуда взялось загадочное 0.9. Если код и так понятен благодаря хорошим именам, комментарий не нужен вовсе.

Попробуйте сами 2.13. Возьмите свою программу из задания 2.10 и добавьте в неё комментарии: что делает программа целиком и откуда берутся данные.

Собираем всё вместе: поездка на дачу

Напишем программу, в которой пригодится всё из этой главы. Семья Петровых собирается на дачу на машине и хочет знать, во что обойдётся бензин и сколько с каждого, если скинуться поровну.

Листинг 2.12. dacha_trip.py

```
# Стоимость поездки на дачу на машине


distance = float(input("Расстояние до дачи в одну сторону, км: "))

consumption = float(input("Расход бензина, литров на 100 км: "))

fuel_price = float(input("Цена литра бензина, руб.: "))

people = int(input("Сколько человек едет? "))


# Туда и обратно, поэтому расстояние удваиваем

liters = distance * 2 * consumption / 100

cost = liters * fuel_price

per_person = cost / people


print()

print(f"Бензина на дорогу туда и обратно: {liters:.1f} л")

print(f"Стоимость поездки: {cost:.2f} руб.")

print(f"С каждого из {people}: {per_person:.2f} руб.")

print(f"Барсик едет бесплатно: {people >= 1}")
```

```
Расстояние до дачи в одну сторону, км: 68

Расход бензина, литров на 100 км: 8.5

Цена литра бензина, руб.: 62.4

Сколько человек едет? 3


Бензина на дорогу туда и обратно: 11.6 л

Стоимость поездки: 721.34 руб.
```

С каждого из 3: 240.45 руб.

Барсик едет бесплатно: True

Пройдёмся по программе. Четыре вопроса - четыре input, и каждый сразу превращается в число: расстояние, расход и цена бывают дробными, поэтому float, а людей всегда целое число, поэтому int. Потом три вычисления, у каждого своя переменная с понятным именем. Комментарий объясняет неочевидное место: почему расстояние умножается на 2. Вывод собран f-строками с форматами: литры с одним знаком после точки, рубли с двумя.

Последняя строка - шутка, но с умыслом: сравнение people >= 1 даёт логическое значение, и f-строка выводит его как True. Если в машине есть хоть один человек, Барсик едет бесплатно. Если людей ноль, то и поездки не будет, но программа об этом пока не знает: при вводе нуля она упадёт с ошибкой деления на ноль. Защищаться от таких случаев мы научимся в следующей главе.

Попробуйте сами 2.14. Дополните программу: пусть она спрашивает ещё и стоимость платной дороги в одну сторону и добавляет её (туда и обратно) к общей стоимости поездки.

Итоги

Переменная - имя, прикреплённое к значению. Знак = присваивает значение, а не сравнивает; значение переменной можно заменить.

Имена пишут маленькими английскими буквами через подчёркивание (snake_case) и выбирают так, чтобы было понятно, что внутри.

Основные типы: целые числа int, дробные float (через точку!), строки str и логический тип bool со значениями True и False. Узнать тип помогает type.

Арифметика: + - * /, целое деление //, остаток %, степень **. Деление / всегда даёт дробное число; дробные числа бывают неточны в последнем знаке, их округляют функцией round.

f-строки подставляют значения в текст: f"Коту {age} года", а формат :.2f оставляет два знака после точки.

input всегда возвращает строку. Числа из неё получают функциями int и float, обратно в строку превращает str.

Комментарии начинаются с # и объясняют, зачем написан код.

Глава 3. Условия

До сих пор наши программы выполнялись одинаково при любых данных: строка за строкой, сверху вниз. Но жизнь состоит из развилок. Если идёт дождь, берём зонт. Если в холодильнике нет молока, идём в магазин. Если кот орёт в шесть утра, значит, миска пуста. В этой главе программа научится принимать решения.

Если: инструкция if

Начнём с самой простой развилки.

Листинг 3.1. umbrella.py

```
answer = input("На улице дождь? (да/нет) ")

if answer == "да":

print("Возьмите зонт!")

print("И резиновые сапоги, если едете на дачу.")

print("Хорошего дня!")
```

Запустим программу дважды, с разными ответами:

```
На улице дождь? (да/нет) да

Возьмите зонт!

И резиновые сапоги, если едете на дачу.

Хорошего дня!
На улице дождь? (да/нет) нет

Хорошего дня!
```

Разберём вторую строку. Слово if значит "если". За ним идёт условие (condition): answer == "да", то есть "ответ равен строке да". В конце строки стоит двоеточие. Оно обязательно и означает: "дальше идут действия, которые надо выполнить, если условие верно".

Эти действия записаны с отступом в четыре пробела. Строки с одинаковым отступом образуют блок (block). Если условие истинно (True), Python выполняет весь блок. Если ложно (False), пропускает его целиком и идёт дальше. Строка print("Хорошего дня!") написана без отступа, она не входит в блок и выполняется всегда.

Вот зачем в Python отступы: они показывают, какие строки "принадлежат" условию. В других языках для этого ставят фигурные скобки или слова begin и end, а в Python - отступ. Поэтому лишний пробел в начале строки, как мы видели в главе 1, - это ошибка, а не небрежность.

Совет. Отступ в Python - это четыре пробела. VS Code ставит их сам, когда вы нажимаете Enter после строки с двоеточием, а клавиша Tab в нём тоже вставляет четыре пробела. Не смешивайте пробелы и настоящие символы табуляции в одном файле: на вид они одинаковые, а Python их различает.

Если забыть про отступ, Python об этом скажет:

Листинг 3.2. error_no_indent.py

```
answer = "да"

if answer == "да":

print("Возьмите зонт!")
File "C:\pyprojects\error_no_indent.py", line 3
```

```
print("Возьмите зонт!")

^^^^^
```

```
IndentationError: expected an indented block after 'if' statement on line 2
```

"Ожидался блок с отступом после инструкции if в строке 2". Python увидел двоеточие и ждёт, что следующая строка будет с отступом.

Попробуйте сами 3.1. Напишите программу, которая спрашивает температуру на улице и, если она ниже нуля, советует надеть шапку. При любой температуре программа должна в конце пожелать хорошей прогулки.

Иначе: else

Часто нужно сделать одно, если условие верно, и другое, если нет. Для второго случая есть слово else ("иначе").

Листинг 3.3. even_odd.py

```
number = int(input("Введите целое число: "))

if number % 2 == 0:

print(f"{number} - чётное число.")

else:

print(f"{number} - нечётное число.")
```

```
Введите целое число: 14

14 - чётное число.
Введите целое число: 7

7 - нечётное число.
```

Здесь пригодился остаток от деления из главы 2: у чётных чисел остаток от деления на 2 равен нулю. else пишется на том же уровне отступа, что и if, и тоже с двоеточием. Выполнится ровно один из двух блоков: или первый, или второй. Оба сразу или ни одного - не бывает.

Попробуйте сами 3.2. Измените программу так, чтобы она проверяла, делится ли число на 3 без остатка.

Сравнения

Условие - это выражение (expression), которое даёт True или False. Чаще всего это сравнение. Вот все знаки сравнения:

Знак
Значение
Пример
Результат

==
равно
5 == 5
True

!=
не равно
5 != 5
False

<
меньше
3 < 5
True

>
больше
3 > 5
False

<=
меньше или равно
5 <= 5
True

>=
больше или равно
4 >= 5
False

Сравнивать можно не только числа, но и строки. Строки сравниваются по алфавиту, точнее, по номерам букв в таблице символов: буква, которая стоит в таблице раньше, считается меньше.

>>> "арбуз" < "яблоко"

True

>>> "Барсик" == "барсик"

```
False

>>> "Барсик" < "барсик"

True

>>> "10" > "9"

False

>>> 10 > 9

True
```

Обратите внимание на два последних сравнения. Числа 10 и 9 сравниваются как числа, и 10 больше. А строки "10" и "9" сравниваются посимвольно, как слова в словаре: первый символ "1" меньше "9", и дальше Python уже не смотрит. Это частая ловушка: если забыть превратить ввод в число функцией int, программа будет сравнивать строки и выдавать странные ответы.

Ещё одна особенность: большие буквы в таблице символов стоят раньше маленьких, поэтому "Барсик" меньше, чем "барсик". А буква ё стоит вообще отдельно, после я. Как сортировать русские слова правильно, разберёмся в главе 7.

В Python можно писать двойные сравнения, как в математике:

```
>>> age = 35

>>> 18 <= age < 65

True

>>> 0 < age < 18

False
```

18 <= age < 65 читается "возраст от 18 включительно и меньше 65". Во многих других языках так писать нельзя, а в Python можно и удобно.

Попробуйте сами 3.3. Не запуская Python, предскажите результат: "кот" == "кот ", "Яблоко" < "арбуз", 3 < 5 < 4, "100" < "25". Проверьте себя.

Попробуйте сами 3.4. Напишите программу, которая спрашивает пароль и, если он равен мяу123, пишет "Добро пожаловать, хозяин!", а иначе "Неверный пароль. Барсик, это ты?".

Несколько вариантов: elif

Когда вариантов больше двух, на помощь приходит elif (сокращение от else if, "иначе если"). Пусть магазин даёт скидку в зависимости от суммы покупки.

Листинг 3.4. discount.py

```
total = float(input("Сумма покупки, руб.: "))


if total >= 5000:

discount = 10

elif total >= 3000:

discount = 5

elif total >= 1000:

discount = 2

else:

discount = 0


to_pay = total * (100 - discount) / 100

print(f"Скидка: {discount}%")

print(f"К оплате: {to_pay:.2f} руб.")
```

```
Сумма покупки, руб.: 3500

Скидка: 5%

К оплате: 3325.00 руб.
Сумма покупки, руб.: 740

Скидка: 0%

К оплате: 740.00 руб.
```

Python проверяет условия сверху вниз и выполняет блок первого же условия, которое оказалось истинным. Остальные он даже не смотрит. При сумме 3500 первое условие (>= 5000) ложно, второе (>= 3000) истинно, значит, скидка 5%, и проверка заканчивается. Если не подошло ни одно условие, выполняется блок else. Он необязателен: если его нет и ничего не подошло, программа просто ничего не делает.

Порядок условий здесь важен. Посмотрите, что будет, если записать их наоборот:

Листинг 3.5. discount_bad.py

```
total = 7000
```

```
if total >= 1000:

discount = 2

elif total >= 3000:

discount = 5

elif total >= 5000:

discount = 10

else:

discount = 0

print(f"Скидка: {discount}%")
```

Скидка: 2%

Покупатель на 7000 рублей получил скидку 2%, хотя заслужил 10%. Сумма 7000 больше 1000, первое условие истинно, и до остальных дело не дошло. Программа без ошибок, Python ничего не заметил, а покупатель расстроился. Такие ошибки в логике (logic errors) коварнее опечаток: о них никто не сообщает. Поэтому при цепочке elif с границами "больше или равно" начинайте с самой большой границы.

Теперь мы можем исправить программу про кошачьи годы из главы 2: научить её считать возраст котят.

Листинг 3.6. cat_years2.py

```
name = input("Как зовут кота? ")

cat_age = int(input("Сколько ему полных лет? "))


if cat_age <= 0:

print("Котёнку меньше года. Он ещё ребёнок, даже по-человечески.")

elif cat_age == 1:

print(f"{name} по-человечески примерно 15-летний подросток.")

elif cat_age == 2:

print(f"{name} по-человечески примерно 24-летний.")

else:

human_age = 24 + (cat_age - 2) * 4

print(f"По-человечески {name} примерно {human_age}-летний.")
```

Как зовут кота? Пушок

Сколько ему полных лет? 1

Пушок по-человечески примерно 15-летний подросток.
Как зовут кота? Барсик

Сколько ему полных лет? 4

По-человечески Барсик примерно 32-летний.

Попробуйте сами 3.5. Напишите программу, которая переводит баллы за контрольную (от 0 до 100) в оценку: 90 и больше - "5", от 75 - "4", от 60 - "3", меньше 60 - "2".

Попробуйте сами 3.6. Високосный год - тот, что делится на 4, но не делится на 100. Исключение: годы, которые делятся на 400, тоже високосные (поэтому 2000 год был високосным, а 1900 - нет). Пока напишите программу с цепочкой elif, а после следующего раздела попробуйте уложить проверку в одно условие.

Логические операторы: and, or, not

Иногда решение зависит сразу от нескольких вещей. "Поливать огород вечером, если было жарко и не было дождя". Для таких случаев есть логические операторы (logical operators):

and ("и") - истинно, если истинны оба условия;
or ("или") - истинно, если истинно хотя бы одно;
not ("не") - переворачивает значение: not True даёт False.

Листинг 3.7. garden.py

```
temperature = int(input("Какая сегодня температура днём? "))

rain = input("Был дождь? (да/нет) ")

frost = input("На ночь обещают заморозки? (да/нет) ")


if temperature > 25 and rain != "да":

print("Жарко и сухо: вечером полейте огород.")


if frost == "да" or temperature < 5:

print("Холодно: укройте рассаду плёнкой.")


if not (rain == "да" or frost == "да"):

print("Без дождя и заморозков. Барсик может гулять во дворе.")
```

```
Какая сегодня температура днём? 28

Был дождь? (да/нет) нет

На ночь обещают заморозки? (да/нет) нет

Жарко и сухо: вечером полейте огород.

Без дождя и заморозков. Барсик может гулять во дворе.
Какая сегодня температура днём? 4

Был дождь? (да/нет) да

На ночь обещают заморозки? (да/нет) нет

Холодно: укройте рассаду плёнкой.
```

Здесь три отдельных if, а не цепочка с elif: советы не исключают друг друга, и могут сработать сразу несколько. Цепочка if/elif/else выбирает один вариант из многих, а несколько отдельных if проверяются каждый сам по себе.

В последнем условии стоят скобки: not (rain == "да" or frost == "да"). Без скобок not относился бы только к первому сравнению. Правило простое: если сомневаетесь в порядке, ставьте скобки. Они ничего не портят, а читать код становится легче.

Что Python считает истиной

В условии может стоять не только сравнение, но и любое значение. Python решает, считать его истинным или ложным, по простому правилу: пустое и нулевое - ложь, всё остальное - истина.

```
>>> bool(0)

False

>>> bool(42)

True

>>> bool("")

False

>>> bool("0")

True

>>> bool("нет")

True
```

Ложны число 0 и пустая строка "". А строка "0" не пустая (в ней есть символ), поэтому истинна. Так же истинна и строка "нет": для Python это просто три буквы, а не отказ.

Это правило удобно, чтобы проверить, ввёл ли человек хоть что-нибудь:

Листинг 3.8. empty_name.py

```
name = input("Как зовут вашего кота? ")

if not name:

print("Вы ничего не ввели. Неужели кот безымянный?")

else:

print(f"{name} - прекрасное имя!")
```

```
Как зовут вашего кота?

Вы ничего не ввели. Неужели кот безымянный?
```

not name истинно, когда строка пустая: человек просто нажал Enter. Запись if not name: короче, чем if name == "":, и программисты на Python пишут именно так.

Попробуйте сами 3.7. Перепишите проверку високосного года из задания 3.6 одним условием с and, or и not. Проверьте на годах 2024, 2025, 1900 и 2000.

Попробуйте сами 3.8. Магазин даёт бесплатную доставку, если сумма заказа от 3000 рублей или если у покупателя есть карта постоянного клиента. Напишите программу, которая спрашивает сумму и есть ли карта, и сообщает, бесплатна ли доставка.

Частые ошибки в условиях

Две ошибки в условиях встречаются у новичков так часто, что заслуживают отдельного разговора.

Первая: одинарное = вместо двойного ==. Рука привыкла к школьному знаку равенства, и получается так:

Листинг 3.9. error_assign_in_if.py

```
answer = input("Хотите сосиску? ")

if answer = "да":

print("Держите сосиску!")
File "C:\pyprojects\error_assign_in_if.py", line 2

if answer = "да":

^^^^^^^^^^^^

SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Python не только нашёл ошибку, но и подсказал: "Может быть, вы имели в виду '==' или ':=' вместо '='?". Нам нужно ==. Про загадочный знак := можно пока не думать, он для более сложных случаев.

Вторая ошибка коварнее, потому что Python о ней молчит. Допустим, мы хотим принять ответ и с маленькой, и с большой буквы и пишем так, как сказали бы словами: "если ответ да или Да".

Листинг 3.10. or_bug.py

```
answer = input("Хотите сосиску? ")

if answer == "да" or "Да":

print("Держите сосиску!")

else:

print("Ну и ладно, Барсик съест.")
Хотите сосиску? нет

Держите сосиску!
```

Человек ответил "нет", а сосиску всё равно получил. Дело в том, как Python читает условие. Для него это (answer == "да") or ("Да"): слева сравнение, справа просто строка "Да". Сравнение дало False, но строка "Да" не пустая, а значит, по правилу из прошлого раздела истинна. False or "Да" - истина, и блок выполняется при любом ответе.

Правильно повторить сравнение в обеих частях:

Листинг 3.11. or_fixed.py

```
answer = input("Хотите сосиску? ")

if answer == "да" or answer == "Да":
```

```
print("Держите сосиску!")

else:

print("Ну и ладно, Барсик съест.")
```

Хотите сосиску? нет

Ну и ладно, Барсик съест.

В главе 7 мы узнаем способ изящнее: перевести ответ в маленькие буквы и сравнивать один раз. Тогда подойдут и "да", и "Да", и даже "ДА", напечатанное с включённым Caps Lock.

Попробуйте сами 3.9. В этой программе ошибка: при вводе 7 она сообщает, что число отрицательное. Найдите и исправьте её.

```
number = int(input("Введите число: "))

if number > 0 or number == 0 or "отрицательное":

print("Число неотрицательное.")

if number < 0 or "отрицательное":

print("Число отрицательное.")
```

Вложенные условия

Внутри блока if может быть другой if. Такие условия называют вложенными (nested). Отступы при этом складываются: блок внутреннего условия сдвинут на восемь пробелов.

Листинг 3.12. cinema.py

```
age = int(input("Сколько вам лет? "))


if age >= 16:

print("Проходите, приятного просмотра!")

else:

with_parent = input("Вы со взрослым? (да/нет) ")

if with_parent == "да":

print("Проходите вместе со взрослым.")

else:

print("Извините, фильм 16+. Приходите с родителями.")
Сколько вам лет? 14

Вы со взрослым? (да/нет) да

Проходите вместе со взрослым.
Сколько вам лет? 12

Вы со взрослым? (да/нет) нет

Извините, фильм 16+. Приходите с родителями.
```

Вопрос про взрослого программа задаёт только тем, кому меньше 16: для остальных он не нужен. Это главное, для чего нужны вложенные условия: второй вопрос имеет смысл только при определённом ответе на первый.

Слишком глубокая вложенность читается плохо: когда отступы уходят далеко вправо, легко запутаться, какой else к какому if относится. Если можно обойтись цепочкой elif или логическими операторами, лучше обойтись. Программисты даже шутят, что код с пятью уровнями вложенности похож на лестницу в подвал, где водятся баги (bugs, так программисты называют ошибки).

Попробуйте сами 3.10. Напишите программу, которая считает цену билета в музей. Полная цена 500 рублей. Детям до 7 лет бесплатно, школьникам до 18 лет скидка 50%, пенсионерам от 65 лет скидка 30%. Если человеку от 18 до 64 лет, программа спрашивает, студент ли он, и даёт студентам скидку 20%.

Условие в одну строку

Бывает, что от условия зависит всего одно значение. Например, совет для кота: на диету или за сосиской. Можно написать четыре строки с if и else, а можно одну:

Листинг 3.13. cat_diet.py

```
weight = float(input("Сколько весит Барсик, кг? "))

advice = "пора на диету" if weight > 6 else "можно дать сосиску"

print(f"Барсик весит {weight} кг: {advice}.")
```

```
Сколько весит Барсик, кг? 6.5

Барсик весит 6.5 кг: пора на диету.
Сколько весит Барсик, кг? 5.2

Барсик весит 5.2 кг: можно дать сосиску.
```

Запись значение1 if условие else значение2 называется условным выражением (conditional expression). Она читается почти по-русски: "пора на диету, если вес больше шести, иначе можно дать сосиску". Условное выражение даёт одно из двух значений, и его можно сохранить в переменной или вставить прямо в f-строку.

Пользуйтесь им для коротких выборов "одно из двух". Если веток больше или в каждой ветке несколько действий, обычные if, elif и else читаются лучше.

Попробуйте сами 3.11. Перепишите программу even_odd.py (листинг 3.3) так, чтобы слово "чётное" или "нечётное" выбиралось условным выражением, а print в программе был всего один.

Выбор по образцу: match

Когда нужно выбрать действие по значению одной переменной, цепочка elif получается однообразной: if drink == "чай", elif drink == "кофе", elif drink == "какао"... Для таких случаев в Python есть инструкция match (сопоставить) с вариантами case (случай).

Листинг 3.14. coffee_machine.py

```
drink = input("Что будете пить? ")


match drink:

case "чай":

print("Чай с лимоном, 60 руб.")

case "кофе" | "капучино":

print("Кофе с молоком, 120 руб.")

case "какао":

print("Какао с зефирками, 90 руб.")

case _:

print("Такого нет. Есть чай, кофе, капучино и какао.")
```

```
Что будете пить? капучино

Кофе с молоком, 120 руб.
Что будете пить? лимонад

Такого нет. Есть чай, кофе, капучино и какао.
```

match drink: значит "сопоставим значение drink с образцами". Python по очереди сравнивает его с образцом (pattern) после каждого case и выполняет блок первого подходящего. Вертикальная черта | означает "или": case "кофе" | "капучино" подходит для обоих слов. Образец _ (подчёркивание) подходит для чего угодно, это аналог else, и его ставят последним.

Пока match выглядит как короткая запись для цепочки elif. Но он умеет больше: разбирать значение на части. Покажем на примере программы, которая понимает команды для списка покупок. Запись command.split() разбивает строку на отдельные слова (подробнее о ней - в главе 7), а match смотрит, сколько получилось слов и какие они.

Листинг 3.15. shopping_commands.py

```
command = input("Команда: ")


match command.split():

case ["список"]:
```

```
        print("Показываю список покупок.")

    case ["купить", item]:

        print(f"Добавлено в список: {item}")

    case ["купить", count, item]:

        print(f"Добавлено в список: {count} {item}")

    case ["выход"] | ["стоп"]:

        print("До свидания!")

    case _:

        print("Не понимаю. Команды: список, купить, выход.")
```

```
Команда: купить молоко

Добавлено в список: молоко
Команда: купить 2 батона

Добавлено в список: 2 батона
Команда: съесть сосиску

Не понимаю. Команды: список, купить, выход.
```

Образец ["купить", item] подходит, если команда состоит ровно из двух слов и первое из них "купить". Второе слово, каким бы оно ни было, попадает в переменную item. Образец ["купить", count, item] - для трёх слов. А команда Барсика "съесть сосиску" не подошла ни под один образец, кроме _.

Без match такую программу пришлось бы писать с проверками длины и сравнением отдельных слов, и она вышла бы вдвое длиннее. В книге мы ещё встретим match в проектах: в "Змейке" (глава 19) он выбирает, что делать в каждом состоянии игры, а в семейном бюджете (глава 22) - какую команду выполнить.

Попробуйте сами 3.12. Напишите программу, которая спрашивает номер дня недели (от 1 до 7) и выводит его название. Для 6 и 7 пусть она добавляет "Выходной!". Используйте match, а неправильный номер обработайте образцом _.

Попробуйте сами 3.13. Дополните листинг 3.15 командой удалить <товар>, которая выводит "Удалено из списка: <товар>", и командой помощь, которая перечисляет все команды.

Собираем всё вместе: викторина

Напишем программу, в которой встретится почти всё из этой главы: викторину про кошек. Она задаёт три вопроса, считает правильные ответы и в конце ставит оценку.

Листинг 3.16. cat_quiz.py

```
print("Викторина: хорошо ли вы знаете кошек?")

score = 0


answer = input("1. Сколько часов в сутки спит кошка: 5, 10 или 15? ")

if answer == "15":

print("Верно! Кошки спят 12-16 часов в сутки.")

score += 1

else:

print("Нет, около 15. Кошки - чемпионы по сну.")


answer = input("2. Чувствуют ли кошки сладкий вкус? (да/нет) ")

if answer == "нет" or answer == "Нет":

print("Верно! Сладкого вкуса кошки не чувствуют.")

score += 1

else:

print("Нет, сладкого они не чувствуют. Сосиски - другое дело.")


answer = input("3. Видит ли кошка в полной темноте? (да/нет) ")

if answer == "нет" or answer == "Нет":

print("Верно! Ей нужно хоть немного света, но совсем немного.")

score += 1

else:

print("Нет, в полной темноте кошка тоже ничего не видит.")
```

```
print()

print(f"Правильных ответов: {score} из 3.")

match score:

case 3:

print("Вы настоящий знаток кошек!")

case 2:

print("Неплохо! Барсик вами доволен.")

case _:

print("Барсик смотрит с укором. Попробуйте ещё раз.")
```

```
Викторина: хорошо ли вы знаете кошек?

1. Сколько часов в сутки спит кошка: 5, 10 или 15? 15

Верно! Кошки спят 12-16 часов в сутки.

2. Чувствуют ли кошки сладкий вкус? (да/нет) да

Нет, сладкого они не чувствуют. Сосиски - другое дело.

3. Видит ли кошка в полной темноте? (да/нет) нет

Верно! Ей нужно хоть немного света, но совсем немного.


Правильных ответов: 2 из 3.

Неплохо! Барсик вами доволен.
```

Каждый вопрос устроен одинаково: спросить, сравнить ответ, похвалить или поправить, при верном ответе прибавить балл. Переменная score копит баллы, а в конце match выбирает оценку по их числу. Образцом в case может быть не только строка, но и число.

Вы наверняка заметили, что три куска программы почти одинаковые. Программисты такого не любят: если захочется задать десять вопросов, программа разрастётся, а исправлять одно и то же придётся в десяти местах. В следующих главах мы узнаем, как избавиться от повторов: циклы (глава 4), списки (глава 5) и функции (глава 8) превратят эту викторину в аккуратную программу на любое число вопросов.

Попробуйте сами 3.14. Добавьте в викторину четвёртый вопрос и поправьте оценки: "знаток" - за 4 правильных ответа, "неплохо" - за 3, остальное - "Барсик смотрит с укором".

Итоги

if условие: выполняет блок с отступом, только если условие истинно. Блок - это строки с одинаковым отступом в четыре пробела.

else: задаёт действия на случай, когда условие ложно, а elif добавляет промежуточные варианты. Выполняется только первый подходящий блок, поэтому порядок условий важен.

Сравнения: == != < > <= >=; строки сравниваются по алфавиту, точнее, по номерам символов. Число 10 больше 9, а строка "10" меньше строки "9".

Логические операторы and, or, not объединяют условия; скобки помогают не запутаться.

Ноль и пустая строка считаются ложью, всё остальное - истиной: if not name: проверяет, что ничего не введено. Поэтому answer == "да" or "Да" истинно всегда: сравнение надо повторять в обеих частях.

Условное выражение выбирает одно из двух значений в одну строку: "чётное" if n % 2 == 0 else "нечётное".

Вложенные условия нужны, когда следующий вопрос имеет смысл только при определённом ответе на предыдущий.

match выбирает действие по образцу: case "чай":, case "кофе" | "капучино":, case _:, а образцы вида ["купить", item] разбирают значение на части.

Глава 4. Циклы

Компьютеры обожают повторять. Человеку надоест выписывать сто строк таблицы, проверять тысячу паролей или в сотый раз говорить коту "Барсик, слезь со стола". Компьютер сделает это миллион раз и не пожалуется. Для повторения в языках программирования есть циклы (loops). В этой главе вы познакомитесь с двумя циклами Python, научитесь вовремя из них выходить и напишете первую настоящую игру.

Цикл while

Слово while переводится как "пока". Цикл while повторяет блок кода, пока условие истинно. Устроим обратный отсчёт перед Новым годом.

Листинг 4.1. countdown.py

```
seconds = 5

while seconds > 0:

print(seconds)

seconds -= 1

print("С Новым годом!")
```

```
5

4

3

2

1

С Новым годом!
```

Запись похожа на if: слово while, условие, двоеточие и блок с отступом. Но if выполняет блок один раз, а while возвращается к условию снова и снова. Порядок такой:

1. Проверить условие seconds > 0. Сейчас seconds равно 5, условие истинно.
2. Выполнить блок: вывести 5, уменьшить seconds до 4.
3. Вернуться к шагу 1.

Каждое такое повторение называется итерацией (iteration). Когда seconds становится равным нулю, условие ложно, цикл заканчивается, и Python переходит к строке после блока: поздравлению.

Строка seconds -= 1 здесь самая важная. Уберите её, и seconds навсегда останется пятёркой, условие всегда будет истинным, а программа будет выводить пятёрки, пока не выключат электричество. Такой цикл называется бесконечным (infinite loop). Если вы случайно его написали, не паникуйте: щёлкните в окне терминала и нажмите Ctrl+C. Python прервёт программу и напишет KeyboardInterrupt - "прервано с клавиатуры".

Частая ошибка. Бесконечный цикл почти всегда означает, что внутри цикла забыли изменить переменную из условия. Если программа "зависла" и ничего не делает или выводит одно и то же, первым делом проверьте, что в условии while и что меняется в блоке.

Цикл с вопросами

Цикл while хорош, когда заранее неизвестно, сколько будет повторений. Например, сколько раз человек ошибётся с паролем.

Листинг 4.2. password_attempts.py

```
secret = "мяу123"
```

```
attempts = 0

password = ""


while password != secret and attempts < 3:

password = input("Пароль: ")

attempts += 1


if password == secret:

print("Добро пожаловать!")

else:

print("Три ошибки подряд. Доступ заблокирован.")
```

Пароль: мяу

Пароль: мур123

Пароль: мяу123

Добро пожаловать!

Цикл крутится, пока пароль неверный и попыток было меньше трёх. Перед циклом переменной password присвоена пустая строка: условие проверяется до первой итерации, и переменная к этому моменту уже должна существовать. Пустая строка точно не совпадёт с секретом, поэтому первый вопрос будет задан обязательно.

Копим на велосипед

Ещё пример, где число повторений заранее неизвестно. Дима копит на велосипед за 25 000 рублей. Каждый месяц он откладывает 3000 рублей на банковский вклад, и банк начисляет 1% в месяц на всё, что уже лежит на вкладе. Через сколько месяцев хватит денег?

Листинг 4.3. bike_savings.py

```
goal = 25000

monthly = 3000

savings = 0

months = 0


while savings < goal:
```

```
savings = savings * 1.01 + monthly

months += 1

print(f"Месяц {months}: на вкладе {savings:.2f} руб.")


print(f"Через {months} мес. велосипед будет куплен!")
```

```
Месяц 1: на вкладе 3000.00 руб.

Месяц 2: на вкладе 6030.00 руб.

Месяц 3: на вкладе 9090.30 руб.

Месяц 4: на вкладе 12181.20 руб.

Месяц 5: на вкладе 15303.02 руб.

Месяц 6: на вкладе 18456.05 руб.

Месяц 7: на вкладе 21640.61 руб.

Месяц 8: на вкладе 24857.01 руб.

Месяц 9: на вкладе 28105.58 руб.

Через 9 мес. велосипед будет куплен!
```

Цикл остановился сам, как только накопления сравнялись с целью или превысили её. Посчитать это без цикла, одной формулой, тоже можно, но формула со сложными процентами понятна далеко не каждому. А цикл просто проживает месяц за месяцем, как сам Дима.

Попробуйте сами 4.1. Сделайте новогодний отсчёт от 10 до 1, а после него выведите "С Новым годом!" три раза. Для трёх поздравлений тоже используйте цикл.

Попробуйте сами 4.2. Напишите программу, которая спрашивает числа одно за другим и складывает их, пока человек не введёт 0. Затем она выводит сумму.

Попробуйте сами 4.3. На пруду растёт кувшинка, и каждый день её листья занимают вдвое большую площадь. Сегодня они занимают 1 квадратный метр, а весь пруд - 500 квадратных метров. Через сколько дней кувшинка закроет весь пруд?

Цикл for и функция range

Второй цикл, for, перебирает элементы по очереди. Самое простое, что можно перебрать, - буквы строки:

Листинг 4.4. letters.py

```
for letter in "Барсик":

print(letter)
```

```
Б

а

р

с

и

к
```

Читается почти по-английски: "для каждой буквы в строке Барсик - вывести букву". Переменная letter на каждой итерации получает следующий символ строки, а когда символы заканчиваются, заканчивается и цикл. Имя переменной выбираем сами: letter, char, x - лишь бы было понятно.

Чаще всего for перебирает числа. Для этого есть функция range (диапазон). Она создаёт последовательность целых чисел:

range(5) - числа от 0 до 4 (пять штук, начиная с нуля);
range(1, 6) - от 1 до 5: начало входит, конец не входит;
range(0, 20, 5) - от 0 до 20 с шагом 5: 0, 5, 10, 15;
range(10, 0, -1) - от 10 до 1 в обратную сторону.

То, что конец не входит в диапазон, поначалу сбивает с толку. Но правило простое: range(1, 6) содержит 6 - 1 = 5 чисел. Чтобы получить числа от 1 до 10, пишут range(1, 11).

Посмотрим на диапазоны в деле. Чтобы числа выводились в одну строку, а не столбиком, у print есть параметр end: он задаёт, что вывести в конце вместо перевода строки. По умолчанию это перевод строки, а мы попросим пробел.

Листинг 4.5. ranges.py

```
for i in range(5):

print(i, end=" ")

print()


for i in range(1, 6):

print(i, end=" ")

print()
```

```
for i in range(0, 20, 5):

print(i, end=" ")

print()


for i in range(10, 0, -1):

print(i, end=" ")

print()
```

0 1 2 3 4

1 2 3 4 5

0 5 10 15

10 9 8 7 6 5 4 3 2 1

Пустой print() после каждого цикла переводит строку, чтобы следующий диапазон начался с новой строки.

Частая ошибка. Ошибка на единицу (off-by-one error) - одна из самых популярных в программировании, ею грешат и профессионалы. Хотели числа от 1 до 10, написали range(1, 10) - получили от 1 до 9. Хотели десять повторений, написали while count <= 10 при count = 0 - получили одиннадцать. Когда пишете границы цикла, проверьте в уме крайние значения: что будет на первой итерации и что на последней.

Таблица умножения

Классика жанра - строка таблицы умножения.

Листинг 4.6. times_table.py

```
number = int(input("Какое число умножаем? "))

for i in range(1, 11):

print(f"{number} x {i} = {number * i}")
```

Какое число умножаем? 7

7 x 1 = 7

7 x 2 = 14

7 x 3 = 21

7 x 4 = 28

7 x 5 = 35

```
7 x 6 = 42

7 x 7 = 49

7 x 8 = 56

7 x 9 = 63

7 x 10 = 70
```

Три строки кода вместо десяти. А если понадобится таблица до ста, изменится одно число.

Накопление суммы

Частый приём: завести переменную до цикла и на каждой итерации что-то к ней прибавлять. Говорят, что маленький Карл Гаусс, будущий великий математик, однажды за минуту сложил все числа от 1 до 100, пока учитель надеялся на час тишины в классе. Гаусс придумал хитрую формулу, а мы просто поручим работу циклу:

Листинг 4.7. sum_to_100.py

```
total = 0

for number in range(1, 101):

total += number

print(f"Сумма чисел от 1 до 100: {total}")
```

```
Сумма чисел от 1 до 100: 5050
```

Переменную, в которой копится результат, называют накопителем (accumulator). Её обязательно нужно создать до цикла и задать начальное значение: для суммы это 0. Если создать её внутри цикла, она будет обнуляться на каждой итерации.

Похожим образом считают количество: заводят счётчик (counter), равный нулю, и прибавляют единицу, когда встречается нужное.

Листинг 4.8. count_letters.py

```
text = "Барсик съел сосиску, а сардельку оставил на завтра"

count = 0

for char in text:

if char == "с":

count += 1

print(f"Буква с встречается {count} раз")
```

```
Буква с встречается 7 раз
```

Внутри цикла стоит условие, и счётчик увеличивается, только если символ совпал. Цикл и условие вместе - самая частая связка в программировании.

Совет. Когда в цикле что-то не так, добавьте внутрь print с переменными: print(f"{char=} {count=}"). Вы увидите, как меняются значения на каждой итерации. Когда ошибка найдена, этот print удаляют.

Попробуйте сами 4.4. Посчитайте сумму всех чётных чисел от 1 до 100. Сделайте это двумя способами: с условием внутри цикла и с шагом в range.

Попробуйте сами 4.5. Факториал числа n (пишется n!) - это произведение всех чисел от 1 до n: 5! = 1 * 2 * 3 * 4 * 5 = 120. Напишите программу, которая спрашивает n и считает его факториал. Подумайте, каким должно быть начальное значение накопителя для произведения.

Попробуйте сами 4.6. Спросите у человека фразу и посчитайте, сколько в ней пробелов. Сколько тогда в ней слов, если слова разделены ровно одним пробелом?

Какой цикл выбрать

Почти любую задачу можно решить и через while, и через for, но обычно один из них подходит лучше. Простое правило:

если известно, сколько раз повторять или что перебирать (десять раз, все буквы строки, числа от 1 до 100), берите for;

если повторять нужно, пока что-то не случится (пока пароль неверный, пока не накопили, пока человек не ввёл "стоп"), берите while.

Сравните, как выглядит один и тот же отсчёт:

Листинг 4.9. two_loops.py

```
print("Через while:")

n = 3

while n > 0:

print(n)

n -= 1


print("Через for:")

for n in range(3, 0, -1):

print(n)
```

```
Через while:

3

2

1

Через for:

3
```

2

1

С for короче и надёжнее: переменная меняется сама, и забыть её уменьшить невозможно, а значит, и бесконечного цикла не будет.

Выход из цикла: break и continue

Иногда цикл нужно прервать раньше времени. Для этого есть инструкция break (прервать). Она немедленно завершает цикл, и Python переходит к строке после него.

Особенно часто break используют вместе с while True. Условие True истинно всегда, поэтому такой цикл бесконечен, пока его не прервёт break. Это удобный способ сказать: "повторяй, пока не случится вот это".

Листинг 4.10. shopping_list.py

```
print("Вводите покупки по одной. Чтобы закончить, введите стоп.")

count = 0

shopping = ""


while True:

item = input("Покупка: ")

if item == "стоп":

break

if item == "":

print("Пустую строку пропускаю.")

continue

count += 1

shopping += f"{count}. {item}\n"


print(f"\nВ списке {count} покупки:")

print(shopping)
```

```
Вводите покупки по одной. Чтобы закончить, введите стоп.

Покупка: молоко

Покупка:

Пустую строку пропускаю.

Покупка: хлеб

Покупка: корм для Барсика
```

Покупка: стоп

В списке 3 покупки:

1. молоко

2. хлеб

3. корм для Барсика

Здесь работает и вторая инструкция, continue (продолжить). Она прерывает не весь цикл, а только текущую итерацию: всё, что ниже неё в блоке, пропускается, и цикл сразу переходит к следующему повтору. Пустую строку мы не считаем покупкой, поэтому для неё continue перескакивает через count += 1.

Разница между ними, если коротко: break - "всё, хватит", continue - "это пропусти, давай следующее".

Обратите внимание, как собирается список. Переменная shopping - строка, к которой на каждой итерации прибавляется новая строчка с переводом строки \n в конце. Это тоже накопитель, только не для числа, а для текста. В следующей главе мы узнаем, как хранить такие списки по-настоящему удобно.

Цикл с else

У циклов в Python есть редкая особенность: блок else. Он выполняется, если цикл закончился сам, а не был прерван break. Звучит странно, но есть задача, где это ровно то, что нужно: поиск. Проверим, простое ли число. Простое число делится только на 1 и на себя: 2, 3, 5, 7, 11, 13 и так далее.

Листинг 4.11. prime.py

```
number = int(input("Введите число больше 1: "))


for divisor in range(2, number):

if number % divisor == 0:

print(f"{number} - составное: делится на {divisor}")

break

else:

print(f"{number} - простое число")
```

```
Введите число больше 1: 91

91 - составное: делится на 7
Введите число больше 1: 97

97 - простое число
```

Цикл перебирает возможные делители. Как только нашёлся делитель, программа сообщает об этом и прерывает цикл, и тогда else не выполняется. Если же цикл дошёл до конца и делитель не нашёлся, срабатывает else: число простое. Обратите внимание на отступ: else стоит на одном уровне с for, а не с if. Это else цикла, а не условия.

Попробуйте сами 4.7. Перепишите программу из задания 4.2 через while True и break: числа складываются, пока человек не введёт слово "стоп". Пустые строки пропускайте с помощью continue.

Попробуйте сами 4.8. Найдите первое число больше 1000, которое делится и на 17, и на 23. Используйте while True или for с большим range и break.

Вложенные циклы

Внутри цикла может быть другой цикл. На каждую итерацию внешнего цикла внутренний проходит целиком. Так устроены часы: пока минутная стрелка делает один шаг, секундная обходит весь круг. Построим полную таблицу умножения.

Листинг 4.12. multiplication_table.py

```
for row in range(1, 10):

for col in range(1, 10):

print(f"{row * col:4}", end="")

print()
```

```
1 2 3 4 5 6 7 8 9

2 4 6 8 10 12 14 16 18

3 6 9 12 15 18 21 24 27

4 8 12 16 20 24 28 32 36

5 10 15 20 25 30 35 40 45

6 12 18 24 30 36 42 48 54

7 14 21 28 35 42 49 56 63

8 16 24 32 40 48 56 64 72

9 18 27 36 45 54 63 72 81
```

Внешний цикл перебирает строки таблицы, внутренний - столбцы. Для каждой строки внутренний цикл выводит девять чисел подряд, без перевода строки (end=""), а пустой print() после него переводит строку. Запись {row * col:4} в f-строке значит "занять ровно 4 позиции". Числа прижимаются к правому краю, и столбцы получаются ровными, хоть в числе одна цифра, хоть две.

Циклы помогают и рисовать. Помните ёлку из главы 1? Её можно построить любой высоты. Вместо внутреннего цикла здесь пригодится умножение строки на число: " " * 3 - это три пробела, а "*" * 5 - пять звёздочек.

Листинг 4.13. yolka_loop.py

```
height = int(input("Высота ёлки: "))


for level in range(1, height + 1):

spaces = " " * (height - level)

stars = "*" * (2 * level - 1)
```

```
print(spaces + stars)


for _ in range(2):

print(" " * (height - 2) + "| |")
```

Высота ёлки: 6

```
*

***

*****

*******

*********

***********

| |

| |
```

На уровне 1 одна звёздочка, на уровне 2 три, на уровне 3 пять: каждый раз на две больше, отсюда 2 * level - 1. А пробелов слева с каждым уровнем на один меньше, поэтому ёлка выходит ровной.

Во втором цикле переменная названа _. Так программисты показывают, что сама переменная не нужна: цикл просто должен выполниться дважды, и номер итерации внутри не используется.

Попробуйте сами 4.9. Спросите у человека ширину и высоту и нарисуйте прямоугольник из символов # нужного размера. Сначала сделайте это вложенными циклами, а потом одним циклом и умножением строки.

Попробуйте сами 4.10. Нарисуйте "лесенку" из цифр, как ниже, для высоты, которую введёт человек. Вот как она выглядит при высоте 4:

```
1

12

123

1234
```

Игра "Угадай число"

Пора собрать всё вместе и написать игру. Компьютер загадывает число от 1 до 100, человек пытается угадать, а компьютер подсказывает: больше или меньше.

Чтобы компьютер мог "загадать", нужен источник случайных чисел. В Python он есть в модуле random. Модуль (module) - это набор готовых функций, которые подключают командой import. Подробно о модулях - в главе 11, а пока достаточно двух строк: import random в начале программы и random.randint(1, 100), когда нужно случайное целое число от 1 до 100 включительно.

Листинг 4.14. guess.py

```
import random


secret = random.randint(1, 100)

attempts = 0

print("Я загадал число от 1 до 100. Попробуйте угадать!")


while True:

guess = int(input("Ваш вариант: "))

attempts += 1

if guess < secret:

print("Моё число больше.")

elif guess > secret:

print("Моё число меньше.")

else:

print(f"Угадали! Это {secret}. Попыток: {attempts}.")

break
```

```
Я загадал число от 1 до 100. Попробуйте угадать!

Ваш вариант: 50

Моё число меньше.

Ваш вариант: 25

Моё число больше.
```

Ваш вариант: 37

Моё число больше.

Ваш вариант: 43

Моё число меньше.

Ваш вариант: 40

Моё число больше.

Ваш вариант: 42

Угадали! Это 42. Попыток: 6.

У вас компьютер загадает другое число: на то они и случайные. Обратите внимание на стратегию игрока. Первый ход - 50, ровно середина. Ответ "больше" или "меньше" сразу отсекает половину вариантов. Следующий ход - середина оставшейся половины, и так далее. Этот способ называется двоичным поиском (binary search). Он гарантирует победу не больше чем за 7 попыток: после каждого хода вариантов остаётся вдвое меньше, а 2 в седьмой степени - это 128, больше ста, поэтому за семь делений пополам из ста вариантов останется один.

Ограничим попытки

Игра станет азартнее, если попыток будет ровно семь. Здесь пригодится цикл for с else: если цикл дошёл до конца без break, значит, попытки кончились.

Листинг 4.15. guess_limited.py

```
import random


secret = random.randint(1, 100)

print("Я загадал число от 1 до 100. У вас 7 попыток.")


for attempt in range(1, 8):

guess = int(input(f"Попытка {attempt}: "))

if guess == secret:

print(f"Угадали с {attempt}-й попытки!")

break

hint = "больше" if guess < secret else "меньше"

print(f"Моё число {hint}.")
```

```
else:

print(f"Попытки кончились. Я загадал {secret}.")
```

Так играет Барсик, который ходит по клавиатуре без всякой стратегии:

```
Я загадал число от 1 до 100. У вас 7 попыток.

Попытка 1: 10

Моё число больше.

Попытка 2: 90

Моё число меньше.

Попытка 3: 50

Моё число меньше.

Попытка 4: 20

Моё число больше.

Попытка 5: 70

Моё число меньше.

Попытка 6: 40

Моё число меньше.

Попытка 7: 60

Моё число меньше.

Попытки кончились. Я загадал 31.
```

Подсказка здесь выбирается условным выражением из главы 3, а номер попытки берётся прямо из переменной цикла attempt: range(1, 8) даёт числа от 1 до 7.

Поменяемся ролями

А теперь пусть угадывает компьютер. Человек загадывает число, а программа применяет двоичный поиск: называет середину отрезка и сужает его по ответам.

Листинг 4.16. computer_guesses.py

```
print("Загадайте число от 1 до 100, а я угадаю.")

print("Отвечайте: больше, меньше или угадал.")
```

```
low = 1

high = 100

attempts = 0


while low <= high:

guess = (low + high) // 2

attempts += 1

answer = input(f"Это {guess}? ")

if answer == "больше":

low = guess + 1

elif answer == "меньше":

high = guess - 1

elif answer == "угадал":

print(f"Ура! Мне понадобилось попыток: {attempts}.")

break

else:

print("Не понимаю. Ответьте: больше, меньше или угадал.")

attempts -= 1

else:

print("Кажется, вы где-то ошиблись с ответом.")
```

Человек загадал 42:
Загадайте число от 1 до 100, а я угадаю.

Отвечайте: больше, меньше или угадал.

Это 50? меньше

Это 25? больше

Это 37? больше

Это 43? меньше

Это 40? больше

Это 41? больше

Это 42? угадал

Ура! Мне понадобилось попыток: 7.

Переменные low и high - границы отрезка, где ещё может прятаться число. Каждый ответ сдвигает одну из границ, и отрезок сужается вдвое. Если человек отвечает невнятно, попытка не засчитывается: attempts -= 1 отменяет только что сделанное увеличение. А если человек запутался и low стал больше high, цикл заканчивается сам, и срабатывает else: такого числа быть не может.

Попробуйте сами 4.11. Добавьте в игру из листинга 4.14 вопрос "Сыграем ещё? (да/нет)" после победы. Для этого всю игру придётся поместить внутрь ещё одного цикла while.

Попробуйте сами 4.12. Пусть игра из задания 4.11 запоминает лучший результат (наименьшее число попыток) за все партии и выводит его при выходе.

Попробуйте сами 4.13. Сыграйте с программой из листинга 4.16, загадав число 1, а потом 100. Сколько понадобилось попыток? Почему никогда не бывает больше семи?

Итоги

Цикл while повторяет блок, пока условие истинно. Внутри цикла должно что-то меняться, иначе цикл станет бесконечным; прервать зависшую программу можно сочетанием Ctrl+C.

Цикл for перебирает элементы по очереди: буквы строки, числа из range. range(1, 11) - это числа от 1 до 10: конец не входит.

Накопитель и счётчик создают до цикла и меняют внутри: так считают суммы, произведения и количество.

break прерывает цикл целиком, continue - только текущую итерацию. while True с break внутри - удобный способ повторять, пока не случится нужное событие.

Блок else у цикла выполняется, если цикл закончился сам, без break; это удобно для поиска.

Во вложенных циклах внутренний цикл проходит целиком на каждой итерации внешнего. Формат {x:4} выравнивает числа в столбцы, а print(..., end="") выводит без перевода строки.

random.randint(1, 100) даёт случайное целое число; двоичный поиск находит число от 1 до 100 не больше чем за 7 попыток.

Глава 5. Списки и кортежи

В прошлой главе мы собирали список покупок в одну длинную строку. Работало, но неуклюже: из такой строки не вычеркнешь хлеб, не посчитаешь, сколько всего покупок, не отсортируешь их по алфавиту. Для хранения многих значений в Python есть специальные типы. Самый главный из них - список. В этой главе вы научитесь создавать списки, менять их, сортировать, перебирать и собирать новые списки из старых одной строкой. А в конце познакомитесь с родственником списка - кортежем.

Что такое список

Список (list) - это упорядоченный набор значений. Записывается он в квадратных скобках, значения разделяются запятыми.

Листинг 5.1. shopping.py

```
shopping = ["молоко", "хлеб", "корм для Барсика", "яблоки"]

prices = [89, 45, 320, 150]

empty = []


print(shopping)

print(prices)

print(empty)

print(f"Покупок в списке: {len(shopping)}")

print(f"Всего к оплате: {sum(prices)} руб.")
```

```
['молоко', 'хлеб', 'корм для Барсика', 'яблоки']

[89, 45, 320, 150]

[]

Покупок в списке: 4

Всего к оплате: 604 руб.
```

Значения в списке называют элементами (elements или items). Элементами могут быть строки, числа, логические значения и даже другие списки. Функция len, знакомая по строкам, показывает, сколько в списке элементов, а функция sum складывает числа из списка. Пустой список [] тоже бывает полезен: с него начинают, когда элементы будут добавляться по ходу программы.

Обратите внимание: при выводе списка целиком Python показывает строки в одинарных кавычках. Так он подчёркивает, что внутри строки, а не числа. Для красивого вывода, как мы скоро увидим, список перебирают в цикле.

Индексы

У каждого элемента списка есть номер - индекс (index). Нумерация начинается с нуля: первый элемент имеет индекс 0, второй - 1, и так далее. Чтобы получить элемент, индекс пишут в квадратных скобках после имени списка.

```
>>> shopping = ["молоко", "хлеб", "корм для Барсика", "яблоки"]

>>> shopping[0]

'молоко'

>>> shopping[2]

'корм для Барсика'

>>> shopping[-1]

'яблоки'

>>> shopping[-2]

'корм для Барсика'
```

Отрицательные индексы считают с конца: -1 - последний элемент, -2 - предпоследний. Это очень удобно: чтобы получить последний элемент, не нужно знать длину списка.

Почему нумерация с нуля? Так исторически сложилось в большинстве языков программирования: индекс показывает, на сколько шагов элемент отстоит от начала. Первый элемент стоит в самом начале, на расстоянии ноль. Поначалу это непривычно, но через неделю вы перестанете это замечать.

Если обратиться к индексу, которого нет, Python сообщит об ошибке:

```
>>> shopping = ["молоко", "хлеб", "корм для Барсика", "яблоки"]

>>> shopping[4]

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

shopping[4]

~~~~~~~~^^^

IndexError: list index out of range
```

"Индекс списка вне диапазона". В списке четыре элемента, их индексы от 0 до 3, а индекса 4 нет. Это частая ошибка на единицу, о которой мы говорили в главе 4.

По индексу элемент можно не только прочитать, но и заменить:

```
>>> shopping = ["молоко", "хлеб", "корм для Барсика"]

>>> shopping[1] = "батон"
```

```
>>> shopping

['молоко', 'батон', 'корм для Барсика']
```

Индексы есть и у строк: "Барсик"[0] - это "Б". А вот заменить букву в строке нельзя. Строки неизменяемы (immutable): однажды созданную строку нельзя поменять, можно только сделать новую.

```
>>> name = "Барсик"

>>> name[0]

'Б'

>>> name[-1]

'к'

>>> name[0] = "М"

Traceback (most recent call last):

File "<python-input-3>", line 1, in <module>

name[0] = "М"

~~~~^^^

TypeError: 'str' object does not support item assignment
```

"Объект str не поддерживает присваивание элементов". Списки же изменяемы (mutable), и в этом их сила.

Попробуйте сами 5.1. Создайте список из пяти дел на выходные. Выведите первое, третье и последнее дело, используя для последнего отрицательный индекс.

Попробуйте сами 5.2. В списке temperatures = [-3, -5, 0, 2, 4, 1, -1] хранится температура за неделю, с понедельника по воскресенье. Выведите температуру в среду и в воскресенье. Исправьте температуру в четверг: на самом деле было 3 градуса.

Срезы

Иногда нужен не один элемент, а кусок списка. Для этого есть срезы (slices). Срез записывается так: список[начало:конец]. Как и в range, начало входит в срез, а конец - нет.

Листинг 5.2. week_weather.py

```
days = ["пн", "вт", "ср", "чт", "пт", "сб", "вс"]

temperatures = [-3, -5, 0, 2, 4, 1, -1]


print("Первые три дня:", days[0:3], temperatures[0:3])

print("Рабочая неделя:", days[:5])

print("Выходные:", days[5:], temperatures[-2:])

print("Через день:", days[::2])

print("Задом наперёд:", days[::-1])
```

```
Первые три дня: ['пн', 'вт', 'ср'] [-3, -5, 0]

Рабочая неделя: ['пн', 'вт', 'ср', 'чт', 'пт']

Выходные: ['сб', 'вс'] [1, -1]

Через день: ['пн', 'ср', 'пт', 'вс']

Задом наперёд: ['вс', 'сб', 'пт', 'чт', 'ср', 'вт', 'пн']
```

Правила такие:

если начало не указано ([:5]), срез идёт с самого начала;

если не указан конец ([5:]), срез идёт до самого конца;

третье число - шаг ([::2] - каждый второй элемент), как в range;

шаг -1 переворачивает список.

Срез всегда создаёт новый список, исходный не меняется. Срезы работают и со строками, поэтому перевернуть слово можно одной записью:

```
>>> "Барсик"[::-1]

'кисраБ'

>>> "шалаш"[::-1]

'шалаш'

>>> "Новый год"[:5]

'Новый'
```

Слово "шалаш" читается одинаково в обе стороны. Такие слова называют палиндромами.

Попробуйте сами 5.3. Напишите программу, которая спрашивает слово и сообщает, палиндром ли оно. Проверьте на словах "казак", "радар" и "кабачок".

Попробуйте сами 5.4. Из списка months = ["январь", "февраль", "март", "апрель", "май", "июнь", "июль", "август", "сентябрь", "октябрь", "ноябрь", "декабрь"] получите срезами: летние месяцы; последний квартал; каждый третий месяц, начиная с марта.

Изменение списка

Списки умеют расти и уменьшаться. Для этого у них есть методы (methods): функции, которые принадлежат значению и вызываются через точку. Самые нужные:

append(x) - добавить элемент в конец;
insert(i, x) - вставить элемент на место с индексом i;
remove(x) - удалить первый элемент, равный x;
pop() - удалить и вернуть последний элемент; pop(i) - элемент с индексом i;
clear() - очистить список.

Листинг 5.3. shopping_edit.py

```
shopping = ["молоко", "хлеб"]

print("Было:", shopping)


shopping.append("сыр")

print("Добавили сыр:", shopping)


shopping.insert(0, "корм для Барсика")

print("Корм - в самое начало:", shopping)


shopping.remove("хлеб")

print("Хлеб есть дома:", shopping)


last = shopping.pop()

print(f"Вычеркнули последнее ({last}):", shopping)


if "молоко" in shopping:

print("Молоко не забыть!")

print("Сыр в списке?", "сыр" in shopping)
```

```
Было: ['молоко', 'хлеб']

Добавили сыр: ['молоко', 'хлеб', 'сыр']

Корм - в самое начало: ['корм для Барсика', 'молоко', 'хлеб', 'сыр']

Хлеб есть дома: ['корм для Барсика', 'молоко', 'сыр']
```

Вычеркнули последнее (сыр): ['корм для Барсика', 'молоко']

Молоко не забыть!

Сыр в списке? False

Оператор in проверяет, есть ли значение в списке, и даёт True или False. Есть и обратный оператор not in. Ещё два полезных метода: index(x) возвращает индекс первого элемента, равного x, а count(x) считает, сколько раз x встречается в списке.

Удалить можно только то, что есть. Попытка удалить отсутствующий элемент - ошибка:

```
>>> shopping = ["молоко", "сыр"]

>>> shopping.remove("колбаса")

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

shopping.remove("колбаса")

~~~~~~~~~~~~~~~^^^^^^^^^^^

ValueError: list.remove(x): x not in list
```

Поэтому перед remove часто проверяют: if "колбаса" in shopping:.

Новое в Python 3.15. Если вы раньше писали на JavaScript, рука может написать push вместо append. Python 3.15 узнаёт такие привычки и подсказывает правильный метод:

```
>>> shopping = ["молоко"]

>>> shopping.push("сыр")

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

shopping.push("сыр")

^^^^^^^^^^^^

AttributeError: 'list' object has no attribute 'push'. Did you mean '.append'?
```

Два имени - один список

Теперь ловушка, в которую попадают все. Папа составил список покупок, а мама решила взять его за основу своего.

Листинг 5.4. two_names.py

```
papa_list = ["хлеб", "молоко"]

mama_list = papa_list

mama_list.append("шоколад")
```

```
print("Мамин список:", mama_list)

print("Папин список:", papa_list)
Мамин список: ['хлеб', 'молоко', 'шоколад']

Папин список: ['хлеб', 'молоко', 'шоколад']
```

Шоколад появился и в папином списке! Вспомните сравнение из главы 2: переменная - это ярлычок на банке. Строка mama_list = papa_list не создала новую банку, а прикрепила второй ярлычок к той же самой. Список один, имён у него два, и изменения видны под любым из них.

Чтобы получить настоящую копию, используйте метод copy():

Листинг 5.5. two_lists.py

```
papa_list = ["хлеб", "молоко"]

mama_list = papa_list.copy()

mama_list.append("шоколад")


print("Мамин список:", mama_list)

print("Папин список:", papa_list)
Мамин список: ['хлеб', 'молоко', 'шоколад']

Папин список: ['хлеб', 'молоко']
```

Со строками и числами такой проблемы нет: они неизменяемы, и "изменить" их можно только создав новое значение, к которому переедет ярлычок. А списки меняются на месте, поэтому важно понимать, сколько у списка ярлычков.

Попробуйте сами 5.5. Начните с пустого списка дел. Добавьте в него три дела методом append, вставьте срочное дело в начало методом insert, удалите одно дело методом remove, а последнее достаньте методом pop и выведите фразу "Сделано: ...". После каждого шага выводите список.

Попробуйте сами 5.6. В списке guests = ["Аня", "Боря", "Вика", "Боря", "Гена"] Боря записан дважды. Посчитайте методом count, сколько раз он встречается, и удалите лишнюю запись так, чтобы Боря остался один.

Сортировка и статистика

Отсортировать список можно двумя способами. Метод sort() сортирует сам список, меняя его. Функция sorted() не трогает исходный список и возвращает новый, отсортированный. Оба по умолчанию сортируют по возрастанию, а с параметром reverse=True - по убыванию.

Листинг 5.6. harvest.py

```
# Урожай помидоров за неделю, кг

harvest = [2.5, 3.1, 1.8, 4.2, 3.6, 2.9, 0.7]


print("По дням:", harvest)

print("По возрастанию:", sorted(harvest))

print("По убыванию:", sorted(harvest, reverse=True))

print("Исходный список не изменился:", harvest)


total = sum(harvest)

print(f"Всего: {total:.1f} кг")

print(f"Лучший день: {max(harvest)} кг, худший: {min(harvest)} кг")

print(f"В среднем: {total / len(harvest):.2f} кг в день")


harvest.sort()

print("После sort():", harvest)
```

```
По дням: [2.5, 3.1, 1.8, 4.2, 3.6, 2.9, 0.7]

По возрастанию: [0.7, 1.8, 2.5, 2.9, 3.1, 3.6, 4.2]

По убыванию: [4.2, 3.6, 3.1, 2.9, 2.5, 1.8, 0.7]

Исходный список не изменился: [2.5, 3.1, 1.8, 4.2, 3.6, 2.9, 0.7]

Всего: 18.8 кг

Лучший день: 4.2 кг, худший: 0.7 кг

В среднем: 2.69 кг в день

После sort(): [0.7, 1.8, 2.5, 2.9, 3.1, 3.6, 4.2]
```

Функции max и min находят наибольший и наименьший элементы, а среднее значение - это сумма, делённая на количество. Пять строк - и у нас статистика урожая, которой позавидует любая соседка по даче.

Частая ошибка. Метод sort() ничего не возвращает: он меняет список на месте. Запись harvest = harvest.sort() - ловушка: в harvest окажется специальное значение None (ничего), а список будет потерян. Нужен новый отсортированный список - пишите sorted(harvest), нужно отсортировать сам список - просто harvest.sort().

Строки сортируются по алфавиту, точнее, по номерам символов, как мы видели в главе 3. Параметр key позволяет сортировать по другому признаку. Например, key=len сортирует строки по длине:

```
>>> pets = ["кот", "черепаха", "хомяк", "попугай"]

>>> sorted(pets)

['кот', 'попугай', 'хомяк', 'черепаха']

>>> sorted(pets, key=len)

['кот', 'хомяк', 'попугай', 'черепаха']
```

Попробуйте сами 5.7. Дан список оценок за четверть: [5, 4, 4, 3, 5, 5, 4, 2, 5]. Выведите среднюю оценку с двумя знаками после точки, лучшую и худшую оценки и сколько раз встречается пятёрка.

Попробуйте сами 5.8. Отсортируйте список имён ["Ольга", "Анна", "Юля", "Борис", "Ян"] по алфавиту, затем в обратном порядке и, наконец, по длине имени.

Перебор списка

Самое частое, что делают со списком, - перебирают его элементы циклом for, точно так же, как буквы строки.

Листинг 5.7. numbered.py

```
shopping = ["молоко", "хлеб", "корм для Барсика"]


for item in shopping:

print("-", item)


print()

for number, item in enumerate(shopping, start=1):

print(f"{number}. {item}")
```

```
- молоко

- хлеб

- корм для Барсика


1. молоко

2. хлеб

3. корм для Барсика
```

Первый цикл просто перебирает покупки. Во втором понадобились номера, и тут помогает функция enumerate (пронумеровать). На каждой итерации она даёт пару: номер и элемент, и обе части сразу раскладываются по двум переменным. Параметр start=1 велит начинать счёт с единицы, как принято у людей, а не с нуля, как принято у списков.

Если есть два списка одинаковой длины, перебрать их одновременно помогает функция zip (застёжка-молния). Она сцепляет списки попарно, как зубчики молнии:

Листинг 5.8. receipt.py

```
products = ["молоко", "хлеб", "сыр", "корм для Барсика"]

prices = [89, 45, 450, 320]


total = 0

for product, price in zip(products, prices):

print(f"{product:<20}{price:>6} руб.")
```

```
total += price

print("-" * 30)

print(f"{'ИТОГО':<20}{total:>6} руб.")
```

молоко 89 руб.

хлеб 45 руб.

сыр 450 руб.

корм для Барсика 320 руб.

ИТОГО 904 руб.

Чек получился ровным благодаря форматам: {product:<20} занимает 20 позиций и прижимает текст влево, а {price:>6} занимает 6 позиций и прижимает число вправо. Подробнее о таких форматах - в главе 7.

Список, который растёт в цикле

Вернёмся к списку покупок из главы 4 и сделаем его по-человечески: каждая покупка - отдельный элемент списка.

Листинг 5.9. shopping_input.py

```
print("Вводите покупки по одной. Чтобы закончить, введите стоп.")

shopping = []


while True:

item = input("Покупка: ")

if item == "стоп":

break

if item in shopping:

print("Это уже есть в списке.")

continue

shopping.append(item)


shopping.sort()
```

```
print(f"\nВ списке покупок {len(shopping)}, по алфавиту:")

for number, item in enumerate(shopping, start=1):

print(f"{number}. {item}")
```

```
Вводите покупки по одной. Чтобы закончить, введите стоп.

Покупка: хлеб

Покупка: молоко

Покупка: хлеб

Это уже есть в списке.

Покупка: яблоки

Покупка: стоп


В списке покупок 3, по алфавиту:

1. молоко

2. хлеб

3. яблоки
```

Список начинается пустым и растёт: append добавляет каждую новую покупку. Проверка item in shopping не даёт записать одно и то же дважды. В конце список сортируется и выводится с номерами. По сравнению с версией из главы 4 программа стала умнее, а длиннее не стала.

Частая ошибка. Не удаляйте элементы из списка внутри цикла for, который этот же список перебирает. Цикл идёт по индексам, и после удаления элементы сдвигаются: следующий элемент "перепрыгивает" на место удалённого, и цикл его пропускает. Если нужно отфильтровать список, создайте новый: как это сделать одной строкой, рассказано в следующем разделе.

Попробуйте сами 5.9. Даны списки names = ["Сергей", "Наташа", "Дима"] и ages = [45, 43, 15]. Выведите с помощью zip строки вида "Сергей - 45 лет". Затем посчитайте средний возраст семьи.

Попробуйте сами 5.10. Напишите программу, которая спрашивает оценки (числа), пока не будет введено слово "стоп", складывает их в список и выводит, сколько оценок введено, среднюю, лучшую и худшую.

Генераторы списков

Очень часто новый список получается из старого: каждый элемент как-то изменить или оставить только подходящие. Это можно сделать циклом:

```
prices = [89, 45, 450, 320]

sale_prices = []

for price in prices:

sale_prices.append(round(price * 0.9, 2))
```

А можно одной строкой с помощью генератора списка (list comprehension):

Листинг 5.10. comprehensions.py

```
prices = [89, 45, 450, 320, 1200]


sale_prices = [round(price * 0.9, 2) for price in prices]

print("Со скидкой 10%:", sale_prices)


expensive = [price for price in prices if price > 300]

print("Дороже 300 рублей:", expensive)


words = ["кот", "собака", "хомяк", "попугай"]

lengths = [len(word) for word in words]

print("Длины слов:", lengths)


squares = [n ** 2 for n in range(1, 11)]

print("Квадраты:", squares)
```

```
Со скидкой 10%: [80.1, 40.5, 405.0, 288.0, 1080.0]

Дороже 300 рублей: [450, 320, 1200]

Длины слов: [3, 6, 5, 7]

Квадраты: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Генератор списка читается почти как фраза: "цена, умноженная на 0.9 и округлённая до копеек, для каждой цены в списке цен". Без round в списке появилось бы число вроде 80.10000000000001: помните сюрприз с дробными числами из главы 2? В квадратных скобках сначала пишут, каким будет новый элемент, потом цикл for, а при необходимости - условие if, которое отбирает элементы. Результат - новый список, исходный не меняется.

Генераторы списков любят за краткость, но злоупотреблять ими не стоит. Если в одну строку приходится втискивать два цикла и три условия, обычный цикл for будет понятнее.

Распаковка в генераторах

Иногда данные лежат в списке списков, а нужен один общий список. Например, траты Петровых записаны по неделям, а посчитать нужно все траты месяца:

Листинг 5.11. flatten.py

```
weeks = [

[1200, 450, 89],

[3200, 150],

[560, 780, 45, 320],

]


all_expenses = [*week for week in weeks]

print("Все траты месяца:", all_expenses)

print("Итого:", sum(all_expenses))

print("Самая крупная трата:", max(all_expenses))
```

```
Все траты месяца: [1200, 450, 89, 3200, 150, 560, 780, 45, 320]

Итого: 6794

Самая крупная трата: 3200
```

Звёздочка перед week значит "выложить все элементы этого списка". Генератор перебирает недели, и из каждой недели в новый список выкладываются все её траты. Получается один плоский список.

Новое в Python 3.15. Звёздочка внутри генератора списка - новинка версии 3.15. В более старых версиях то же самое пишут двумя циклами: [cost for week in weeks for cost in week]. Если ваш код будет запускаться на старом Python, используйте этот вариант.

Попробуйте сами 5.11. Из списка температур [-3, -5, 0, 2, 4, 1, -1] получите генераторами списков: список только морозных дней (ниже нуля) и список температур в градусах Фаренгейта (формула: F = C * 9 / 5 + 32).

Попробуйте сами 5.12. С помощью генератора списков составьте список всех чисел от 1 до 100, которые делятся на 7, и выведите их количество и сумму.

Кортежи

Кортеж (tuple) похож на список, но неизменяем: после создания его нельзя поменять. Записывается кортеж в круглых скобках.

Листинг 5.12. dacha.py

```
dacha = (56.35, 37.52) # широта и долгота

birthday = (12, "марта", 2022) # день рождения Барсика


print("Координаты дачи:", dacha)

print("Широта:", dacha[0])


latitude, longitude = dacha

print(f"Широта {latitude}, долгота {longitude}")


day, month, year = birthday

print(f"Барсик родился {day} {month} {year} года.")
```

```
Координаты дачи: (56.35, 37.52)

Широта: 56.35

Широта 56.35, долгота 37.52

Барсик родился 12 марта 2022 года.
```

Индексы и срезы у кортежей работают так же, как у списков. А в строке latitude, longitude = dacha происходит распаковка (unpacking): элементы кортежа раскладываются по переменным слева. Количество переменных должно совпадать с количеством элементов.

Зачем нужен кортеж, если есть список? Он подходит для данных, которые по смыслу едины и не должны меняться: координаты, дата, цвет из трёх составляющих. Неизменяемость защищает от случайной порчи: кортеж нельзя нечаянно изменить где-то в другой части программы. Если же попытаться, Python 3.15 подскажет, что, возможно, вам нужен список:

```
>>> dacha = (56.35, 37.52)

>>> dacha.append(100)

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

dacha.append(100)

^^^^^^^^^^^
```

AttributeError: 'tuple' object has no attribute 'append'. Did you mean to use a 'list' object?
Распаковка даёт красивый приём - обмен значений двух переменных без третьей:
>>> a = "чай"

>>> b = "кофе"

>>> a, b = b, a

>>> print(a, b)

кофе чай
Справа создаётся кортеж (b, a), а слева он тут же распаковывается в a и b. Скобки вокруг кортежа можно не писать, запятой достаточно. Одна тонкость: кортеж из одного элемента пишется с запятой, (5,), потому что (5) - это просто число 5 в скобках.

Список кортежей

Кортежи часто живут внутри списков. Например, прайс-лист: каждый товар - кортеж из названия и цены.
Листинг 5.13. price_list.py
price_list = [

("молоко", 89),

("хлеб", 45),

("сыр", 450),

("корм для Барсика", 320),

]

for name, price in price_list:

print(f"{name}: {price} руб.")

cheap = [name for name, price in price_list if price < 100]

print("Дешевле 100 рублей:", cheap)
молоко: 89 руб.

хлеб: 45 руб.

сыр: 450 руб.

корм для Барсика: 320 руб.

Дешевле 100 рублей: ['молоко', 'хлеб']

Распаковка работает прямо в заголовке цикла: for name, price in price_list на каждой итерации раскладывает очередной кортеж на название и цену. И в генераторе списка тоже. Как отсортировать такой список по цене, вы узнаете в главе 8, когда познакомитесь с функциями.

Попробуйте сами 5.13. Сохраните в кортеже свой день рождения (день, месяц, год) и выведите его распаковкой в виде "Я родился 5 мая 2010 года".

Попробуйте сами 5.14. Создайте список кортежей "город - население в тысячах" для пяти городов. Выведите города, где живёт больше миллиона человек, и общую численность населения всех пяти.

Собираем всё вместе: викторина на списках

В главе 3 мы написали викторину про кошек, и три вопроса в ней повторяли друг друга почти дословно. Теперь у нас есть списки, кортежи и циклы, и викторину можно сделать аккуратной: вопросы - отдельно, логика - отдельно.

Листинг 5.14. quiz_list.py

```
questions = [

("Сколько часов в сутки спит кошка: 5, 10 или 15?", "15"),

("Чувствуют ли кошки сладкий вкус? (да/нет)", "нет"),

("Видит ли кошка в полной темноте? (да/нет)", "нет"),

("Как называется детёныш кошки?", "котёнок"),

]


print("Викторина: хорошо ли вы знаете кошек?")

score = 0

for number, (question, right_answer) in enumerate(questions, start=1):

answer = input(f"{number}. {question} ")

if answer == right_answer:

print("Верно!")

score += 1

else:

print(f"Нет, правильный ответ: {right_answer}.")


print(f"\nПравильных ответов: {score} из {len(questions)}.")
```

```
Викторина: хорошо ли вы знаете кошек?

1. Сколько часов в сутки спит кошка: 5, 10 или 15? 15

Верно!

2. Чувствуют ли кошки сладкий вкус? (да/нет) да

Нет, правильный ответ: нет.
```

3. Видит ли кошка в полной темноте? (да/нет) нет

Верно!

4. Как называется детёныш кошки? котёнок

Верно!

Правильных ответов: 3 из 4.

Все вопросы с ответами хранятся в списке кортежей. Цикл перебирает их с номерами, и каждый кортеж распаковывается в вопрос и правильный ответ: скобки в (question, right_answer) нужны, потому что enumerate даёт пару "номер и кортеж". Код вопроса написан один раз, а вопросов может быть хоть сто: добавить новый - значит дописать одну строку в список. Именно так и пишут настоящие программы: данные отдельно, логика отдельно.

Попробуйте сами 5.15. Добавьте в викторину ещё два вопроса и сделайте так, чтобы в конце программа выводила процент правильных ответов с точностью до целого.

Итоги

Список - упорядоченный изменяемый набор значений в квадратных скобках. len даёт длину, sum, min, max - сумму, наименьший и наибольший элементы.

Индексы начинаются с нуля, отрицательные считают с конца; обращение к несуществующему индексу даёт IndexError. Срез [начало:конец:шаг] возвращает новый список, конец не входит.

Методы append, insert, remove, pop, clear меняют список, оператор in проверяет наличие элемента. Строки, в отличие от списков, неизменяемы.

Присваивание b = a не копирует список, а даёт ему второе имя; копию делает a.copy().

sort() сортирует сам список и ничего не возвращает, sorted() возвращает новый список; параметры reverse=True и key=len меняют порядок.

enumerate нумерует элементы при переборе, zip перебирает несколько списков сразу.

Генератор списка [выражение for x in список if условие] строит новый список одной строкой; в Python 3.15 в нём можно распаковывать вложенные списки звёздочкой.

Кортеж - неизменяемый список в круглых скобках; распаковка a, b = b, a раскладывает его по переменным.

Глава 6. Словари и множества

Списки хороши, когда важен порядок: первая покупка, вторая, третья. Но часто данные устроены иначе. Цена молока, телефон бабушки, вес Барсика - здесь нужен не номер по порядку, а ответ на вопрос "сколько стоит молоко?" или "какой телефон у бабушки?". Для таких данных в Python есть словари. В этой главе вы научитесь с ними работать, познакомитесь с множествами, которые умеют убирать повторы и сравнивать наборы, и с frozendict - новинкой Python 3.15.

Что такое словарь

Вспомните обычный бумажный словарь: находите слово и рядом читаете перевод. Или записную книжку: находите имя и рядом видите телефон. Словарь (dictionary, коротко dict) в Python устроен так же: он хранит пары "ключ - значение" (key - value). По ключу можно быстро найти значение.

Листинг 6.1. prices.py

```
prices = {

"молоко": 89,

"хлеб": 45,

"сыр": 450,

}


print(prices)

print("Молоко стоит", prices["молоко"], "руб.")


prices["корм для Барсика"] = 320 # новая пара

prices["хлеб"] = 49 # хлеб подорожал

print(prices)


del prices["сыр"]

print("Сыр больше не покупаем:", prices)

print("Товаров в прайсе:", len(prices))

print("Есть ли в прайсе хлеб?", "хлеб" in prices)
```

```
{'молоко': 89, 'хлеб': 45, 'сыр': 450}

Молоко стоит 89 руб.

{'молоко': 89, 'хлеб': 49, 'сыр': 450, 'корм для Барсика': 320}

Сыр больше не покупаем: {'молоко': 89, 'хлеб': 49, 'корм для Барсика': 320}

Товаров в прайсе: 3

Есть ли в прайсе хлеб? True
```

Словарь записывается в фигурных скобках. Каждая пара - это ключ, двоеточие и значение, а пары разделяются запятыми. Длинный словарь удобно записывать столбиком, по паре на строку, как в листинге. Запятая после последней пары не обязательна, но так проще дописывать новые строки.

Значение достают по ключу в квадратных скобках: prices["молоко"]. Так же его и меняют. Если ключа ещё нет, присваивание создаёт новую пару, если есть - заменяет значение. Удаляет пару инструкция del. Функция len показывает, сколько в словаре пар, а оператор in проверяет, есть ли ключ.

Обратите внимание: словарь помнит порядок, в котором добавлялись пары. Новый ключ "корм для Барсика" встал в конец.

Если ключа нет

Если попросить у словаря ключ, которого в нём нет, будет ошибка:

```
>>> prices = {"молоко": 89, "хлеб": 45}

>>> prices["колбаса"]

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

prices["колбаса"]

~~~~~~^^^^^^^^^^^

KeyError: 'колбаса'
```

KeyError - ошибка ключа, и в сообщении указан сам ключ, которого не нашлось. Чтобы обойтись без ошибки, есть метод get. Он возвращает значение, а если ключа нет - специальное значение None ("ничего") или то значение по умолчанию, которое мы укажем вторым аргументом (аргументы - это значения в скобках при вызове, подробно о них - в главе 8):

```
>>> prices = {"молоко": 89, "хлеб": 45}

>>> prices.get("хлеб")

45

>>> print(prices.get("колбаса"))

None

>>> prices.get("колбаса", 0)

0
```

None - особое значение, которое означает отсутствие значения. Интерактивный режим его не показывает, поэтому пришлось вывести через print. Мы ещё встретим None в главе 8, когда будем писать функции.

Напишем маленький русско-английский словарик для поездки за границу.

Листинг 6.2. translator.py

```
words = {

"кот": "cat",

"молоко": "milk",

"спасибо": "thank you",

"сколько стоит": "how much",

}


word = input("Слово по-русски: ")

translation = words.get(word, "не знаю такого слова")

print(f"{word} - {translation}")
Слово по-русски: кот

кот - cat
Слово по-русски: сосиска

сосиска - не знаю такого слова
```

Что может быть ключом

Значения в словаре могут быть любыми: числа, строки, списки, другие словари. А вот ключи должны быть неизменяемыми: строки, числа, кортежи. Список ключом быть не может, потому что он может измениться, и словарь потеряет пару. Ключи в словаре уникальны: если записать одинаковый ключ дважды, останется последнее значение.

Попробуйте сами 6.1. Создайте словарь с возрастом членов семьи Петровых: папе Сергею 45, маме Наташе 43, Диме 15, Барсику 4. Выведите возраст Димы, потом отметьте день рождения Барсика (увеличьте его возраст на 1) и выведите словарь целиком.

Попробуйте сами 6.2. Допишите в словарик из листинга 6.2 ещё пять слов. Сделайте так, чтобы программа переводила слова в цикле, пока человек не введёт пустую строку.

Попробуйте сами 6.3. Предскажите, что выведет программа, а потом проверьте:

```
d = {"a": 1, "b": 2, "a": 3}

print(d, len(d))
```

Перебор словаря

Словарь перебирают циклом for, как список. Но тут есть выбор: перебирать ключи, значения или пары целиком.

Листинг 6.3. expenses.py

```
expenses = {

"продукты": 18500,

"коммуналка": 6800,

"транспорт": 3200,

"кот": 2100,

"развлечения": 4000,

}


for category in expenses:

print(category, end="; ")

print()


total = sum(expenses.values())

print(f"Всего за месяц: {total} руб.")


for category, amount in expenses.items():

share = amount / total * 100

print(f"{category:<12}{amount:>7} руб.{share:>6.1f}%")


biggest = max(expenses, key=expenses.get)

print(f"Больше всего ушло на категорию: {biggest}")
```

```
продукты; коммуналка; транспорт; кот; развлечения;

Всего за месяц: 34600 руб.

продукты 18500 руб. 53.5%
```

коммуналка 6800 руб. 19.7%

транспорт 3200 руб. 9.2%

кот 2100 руб. 6.1%

развлечения 4000 руб. 11.6%

Больше всего ушло на категорию: продукты

Цикл for category in expenses перебирает ключи: просто так словарь отдаёт именно их. Метод values() даёт все значения, и их удобно передать в sum. А метод items() даёт пары "ключ - значение", которые сразу распаковываются в две переменные, как кортежи в главе 5. Есть ещё метод keys() с ключами, но он нужен редко: ключи и так перебираются по умолчанию.

Строка с max - приём, который стоит запомнить. Функция max перебирает ключи словаря, но сравнивает их не сами по себе, а по тому, что вернёт expenses.get для каждого ключа. То есть ищется ключ с наибольшим значением. Похожим образом словарь сортируют по значениям: sorted(expenses, key=expenses.get).

Подсчёт с помощью словаря

Словари часто используют для подсчёта: ключ - то, что считаем, значение - сколько раз встретилось. Посчитаем, какие слова чаще всего встречаются в отзыве покупателя.

Листинг 6.4. word_count.py

```
review = "кот доволен корм вкусный кот ест корм с утра до вечера корм отличный"


counts = {}

for word in review.split():

counts[word] = counts.get(word, 0) + 1


print(counts)

for word in sorted(counts, key=counts.get, reverse=True)[:3]:

print(f"{word}: {counts[word]}")
```

{'кот': 2, 'доволен': 1, 'корм': 3, 'вкусный': 1, 'ест': 1, 'с': 1, 'утра': 1, 'до': 1, 'вечера': 1, 'отличный': 1}

корм: 3

кот: 2

доволен: 1

Главная строка - counts[word] = counts.get(word, 0) + 1. Если слово встречается впервые, get вернёт 0, и в словарь запишется 1. Если слово уже было, get вернёт его счёт, и он увеличится на 1. Потом мы сортируем слова по их счёту в обратном порядке и берём срезом первые три.

Подсчёт встречается в программировании так часто, что в стандартной библиотеке Python для него есть готовый инструмент Counter; мы познакомимся с ним в главе 11.

Попробуйте сами 6.4. Возьмите словарь expenses из листинга 6.3 и выведите категории, отсортированные по сумме от большей к меньшей, с суммами.

Попробуйте сами 6.5. Посчитайте, сколько раз встречается каждая буква в слове "абракадабра". Выведите буквы и их количество.

Попробуйте сами 6.6. Дан словарь оценок grades = {"Аня": 5, "Боря": 3, "Вика": 4, "Гена": 5, "Даша": 4}. Выведите средний балл, имена всех отличников и имя ученика с самой низкой оценкой.

Вложенные структуры

Самое интересное начинается, когда структуры вкладываются друг в друга. Значением словаря может быть список или другой словарь, а элементами списка - словари. Так в программах описывают почти любые данные из жизни.

Список словарей

Записная книжка, где у каждого человека несколько полей: имя, телефон, день рождения. Каждый человек - словарь, а вся книжка - список таких словарей.

Листинг 6.5. contacts.py

```
contacts = [

{"name": "Бабушка", "phone": "+7 900 111-22-33", "birthday": "12.03"},

{"name": "Тётя Валя", "phone": "+7 900 444-55-66", "birthday": "30.07"},

{"name": "Ветеринар", "phone": "+7 495 777-88-99", "birthday": None},

]


for person in contacts:

line = f"{person['name']}: {person['phone']}"

if person["birthday"]:

line += f", день рождения {person['birthday']}"

print(line)


wanted = "Тётя Валя"

for person in contacts:

if person["name"] == wanted:

print(f"Нашла! Звоним по номеру {person['phone']}")

break

else:

print(f"{wanted} в книжке не найдена")
```

```
Бабушка: +7 900 111-22-33, день рождения 12.03
```

Тётя Валя: +7 900 444-55-66, день рождения 30.07

Ветеринар: +7 495 777-88-99

Нашла! Звоним по номеру +7 900 444-55-66

Внутри f-строки ключи словаря написаны в одинарных кавычках: person['name']. Сама f-строка в двойных кавычках, а разные кавычки помогают глазу увидеть, где она кончается. С версии 3.12 Python понимает и двойные кавычки внутри, но в старых версиях это была ошибка, поэтому многие по привычке пишут одинарные. У ветеринара дня рождения в книжке нет, и вместо него стоит None: условие if person["birthday"] считает None ложью, как пустую строку. Поиск по имени сделан циклом с break и else из главы 4.

Словарь списков

На даче несколько грядок, и на каждой растёт несколько культур. Ключ - название грядки, значение - список того, что на ней посажено.

Листинг 6.6. garden_beds.py

```
beds = {

"у забора": ["малина", "смородина"],

"у дома": ["помидоры", "базилик", "перец"],

"за баней": ["картошка"],

}


beds["за баней"].append("кабачки")

beds["новая"] = ["клубника"]


for bed, plants in beds.items():

print(f"Грядка {bed}: {', '.join(plants)}")


plant = "кабачки"

for bed, plants in beds.items():

if plant in plants:

print(f"{plant.capitalize()} растут на грядке {bed}.")
```

Грядка у забора: малина, смородина

Грядка у дома: помидоры, базилик, перец

```
Грядка за баней: картошка, кабачки

Грядка новая: клубника

Кабачки растут на грядке за баней.
```

beds["за баней"] - это список, поэтому к нему сразу можно применить метод append: выражение читается слева направо, сначала достаём список по ключу, потом добавляем в него элемент. Запись ', '.join(plants) склеивает элементы списка в одну строку через запятую с пробелом, а capitalize() делает первую букву строки заглавной. Оба метода строк подробно разберём в главе 7.

Словарь словарей

Наконец, данные о питомцах: у каждого свой набор свойств.

```
>>> pets = {

... "Барсик": {"вид": "кот", "возраст": 4, "вес": 5.6},

... "Кеша": {"вид": "попугай", "возраст": 2, "вес": 0.1},

... }

>>> pets["Барсик"]["вес"]

5.6

>>> pets["Кеша"]["возраст"] += 1

>>> pets["Кеша"]

{'вид': 'попугай', 'возраст': 3, 'вес': 0.1}
```

Квадратные скобки идут цепочкой: pets["Барсик"] даёт словарь Барсика, а ["вес"] достаёт из него вес. Такие вложенные структуры из списков и словарей очень похожи на формат JSON, в котором хранят данные программы и обмениваются ими сайты. С JSON мы будем работать в главе 10.

Попробуйте сами 6.7. Добавьте в книжку из листинга 6.5 ещё два контакта. Выведите имена всех, у кого день рождения летом (месяц 06, 07 или 08). Подсказка: срез строки "30.07"[3:] даёт номер месяца.

Попробуйте сами 6.8. В словаре beds из листинга 6.6 посчитайте, сколько всего культур растёт на даче, и выведите грядку, на которой их больше всего.

Множества

Множество (set) - это набор уникальных значений без определённого порядка. Записывается в фигурных скобках, как словарь, но без двоеточий.

```
>>> numbers = {3, 1, 2, 3, 1}

>>> numbers

{1, 2, 3}

>>> len(numbers)

3

>>> 2 in numbers

True

>>> numbers.add(10)

>>> numbers.discard(1)

>>> numbers

{10, 2, 3}
```

Повторы исчезли сами: во множестве каждое значение встречается не больше одного раза. Метод add добавляет элемент, discard удаляет (и не ругается, если элемента нет), а in проверяет, есть ли элемент, причём во множестве это работает гораздо быстрее, чем в длинном списке.

Пустое множество создают функцией set(). Пустые фигурные скобки {} - это не множество, а пустой словарь: словарь в Python появился раньше и занял эту запись.

Частая ошибка. У множества нет порядка. Если вывести множество строк, порядок элементов может быть любым и даже меняться от запуска к запуску. Не рассчитывайте на него: если нужен порядок, превратите множество в отсортированный список функцией sorted.

Убрать повторы

Самое частое применение множеств - избавиться от дубликатов.

Листинг 6.7. unique.py

```
bought = ["молоко", "хлеб", "молоко", "сыр", "хлеб", "молоко"]

unique = set(bought)

print(f"Покупок было {len(bought)}, разных товаров {len(unique)}")

print("Что покупали:", sorted(unique))
```

```
Покупок было 6, разных товаров 3
```

Что покупали: ['молоко', 'сыр', 'хлеб']

Функция set превращает список во множество, и повторы пропадают. А sorted возвращает отсортированный список, чтобы вывод был аккуратным и одинаковым при каждом запуске.

Операции над множествами

Множества умеют то, что вы, возможно, помните из школьной математики: объединение, пересечение и разность. Петровы составляют список гостей на дачу: у мамы свой список друзей, у папы свой.

Листинг 6.8. guests.py

```
mama = {"Ольга", "Ира", "Андрей", "тётя Валя"}

papa = {"Андрей", "Костя", "Ольга", "дядя Коля"}


print("Все гости:", sorted(mama | papa))

print("Общие друзья:", sorted(mama & papa))

print("Только мамины:", sorted(mama - papa))

print("Только папины:", sorted(papa - mama))

print("Знает только кто-то один:", sorted(mama ^ papa))
```

```
Все гости: ['Андрей', 'Ира', 'Костя', 'Ольга', 'дядя Коля', 'тётя Валя']

Общие друзья: ['Андрей', 'Ольга']

Только мамины: ['Ира', 'тётя Валя']

Только папины: ['Костя', 'дядя Коля']

Знает только кто-то один: ['Ира', 'Костя', 'дядя Коля', 'тётя Валя']
```

| - объединение (union): все элементы из обоих множеств;

& - пересечение (intersection): только те, что есть в обоих;

- - разность (difference): те, что есть в первом, но нет во втором;

^ - симметрическая разность (symmetric difference): те, что есть только в одном из двух.

Без множеств пришлось бы писать вложенные циклы с проверками, а так - одна операция на каждый вопрос.

Попробуйте сами 6.9. Соседи по даче записали, что посадили. У Петровых: помидоры, огурцы, кабачки, укроп, клубника. У соседей: огурцы, картошка, клубника, малина, укроп. Найдите, что посадили и те, и другие, что есть только у Петровых и сколько всего разных культур на двух участках.

Попробуйте сами 6.10. Спросите у человека фразу и выведите, сколько в ней разных букв (без пробелов). Подсказка: множество из строки создаётся так же, как из списка: set("мама").

frozendict - новинка Python 3.15

У списка есть неизменяемый родственник - кортеж. А у словаря такого родственника долго не было. В Python 3.15 он появился: frozendict, "замороженный словарь". Он умеет всё, что словарь, кроме одного: его нельзя изменить после создания.

Листинг 6.9. settings.py

```
DEFAULT_SETTINGS = frozendict(theme="светлая", font_size=12, sound=True)


print(DEFAULT_SETTINGS)

print("Размер шрифта по умолчанию:", DEFAULT_SETTINGS["font_size"])


my_settings = DEFAULT_SETTINGS | {"theme": "тёмная"}

print("Мои настройки:", my_settings)

print("Настройки по умолчанию не изменились:", DEFAULT_SETTINGS)


DEFAULT_SETTINGS["sound"] = False
frozendict({'theme': 'светлая', 'font_size': 12, 'sound': True})

Размер шрифта по умолчанию: 12

Мои настройки: frozendict({'theme': 'тёмная', 'font_size': 12, 'sound': True})

Настройки по умолчанию не изменились: frozendict({'theme': 'светлая', 'font_size': 12, 'sound': True})

Traceback (most recent call last):

File "C:\pyprojects\settings.py", line 10, in <module>

DEFAULT_SETTINGS["sound"] = False

~~~~~~~~~~~~~~~~~^^^^^^^^^

TypeError: 'frozendict' object does not support item assignment
```

frozendict создаётся функцией frozendict. Пары можно передать как именованные аргументы, как здесь, или готовым словарём: frozendict({"theme": "светлая"}). Читать из него можно всеми привычными способами: квадратными скобками, get, items(), циклом for. А вот любая попытка изменить его заканчивается ошибкой TypeError: "объект frozendict не поддерживает присваивание элементов".

Нужны другие значения - создайте новый frozendict на основе старого. Оператор | объединяет словари: справа пары, которые добавляются или заменяют старые. Результат - новый frozendict, а исходный остаётся нетронутым.

Имя DEFAULT_SETTINGS написано большими буквами. Так программисты на Python договорились называть константы (constants) - значения, которые задаются один раз и не меняются во время работы программы. С обычным словарём это была только договорённость: кто-нибудь мог случайно поменять настройки по умолчанию в другом месте программы. С frozendict это невозможно.

frozendict как ключ

У неизменяемости есть второе достоинство. frozendict можно сделать ключом словаря или элементом множества, а обычный словарь - нельзя:

```
>>> orders = set()

>>> orders.add({"блюдо": "борщ", "хлеб": True})

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

orders.add({"блюдо": "борщ", "хлеб": True})

~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

TypeError: cannot use 'dict' as a set element (unhashable type: 'dict')

>>> orders.add(frozendict({"блюдо": "борщ", "хлеб": True}))

>>> frozendict(хлеб=True, блюдо="борщ") in orders

True
```

Python объясняет, что словарь нельзя использовать как элемент множества: он изменяемый (unhashable type - "нехешируемый тип"; хеш - это числовой отпечаток значения, по которому множества и словари быстро находят элементы). А frozendict добавился без проблем. Обратите внимание на последнюю проверку: frozendict с теми же парами, но записанными в другом порядке, равен первому и находится во множестве.

На этом можно построить защиту от повторов. Например, в кафе одинаковые заказы приходят дважды, когда официант дважды нажимает кнопку:

Листинг 6.10. orders.py

```
orders = [

{"стол": 3, "блюдо": "борщ", "хлеб": True},

{"стол": 5, "блюдо": "плов", "хлеб": False},

{"хлеб": True, "блюдо": "борщ", "стол": 3},

]


seen = set()
```

```
for order in orders:

key = frozendict(order)

if key in seen:

print("Повтор, пропускаю:", dict(key))

continue

seen.add(key)

print("Принят заказ:", dict(key))
```

```
Принят заказ: {'стол': 3, 'блюдо': 'борщ', 'хлеб': True}

Принят заказ: {'стол': 5, 'блюдо': 'плов', 'хлеб': False}

Повтор, пропускаю: {'хлеб': True, 'блюдо': 'борщ', 'стол': 3}
```

Третий заказ - тот же самый, что и первый, хоть пары и записаны в другом порядке. Множество seen помнит все принятые заказы, а frozendict делает каждый заказ пригодным для хранения во множестве. Функция dict превращает frozendict обратно в обычный словарь: здесь только ради того, чтобы вывод был покороче.

Новое в Python 3.15. frozendict - встроенный тип, импортировать его не нужно. Модуль json, с которым мы познакомимся в главе 10, умеет сохранять frozendict так же, как обычный словарь. В более старых версиях Python этого типа нет, и программа с ним выдаст NameError.

Попробуйте сами 6.11. Создайте frozendict WORK_HOURS с часами работы магазина по дням недели (например, "пн": "9-21"). Получите на его основе новый frozendict, где в воскресенье магазин работает с 10 до 18, и выведите оба.

Собираем всё вместе: записная книжка

Напишем программу-записную книжку, которая понимает команды: добавить номер, найти, удалить, показать всё. Номера хранятся в словаре "имя - телефон", команды разбирает match из главы 3.

Листинг 6.11. phonebook.py

```
phonebook = {"бабушка": "+7 900 111-22-33"}

print("Записная книжка. Команды: добавить, найти, удалить, все, выход.")


while True:

command = input("> ")

match command:

case "добавить":

name = input("Имя: ")

phone = input("Телефон: ")

phonebook[name] = phone

print(f"Записала: {name}")

case "найти":

name = input("Кого ищем? ")

print(phonebook.get(name, "Такого имени нет"))

case "удалить":

name = input("Кого удалить? ")

if name in phonebook:

del phonebook[name]

print(f"Удалила: {name}")

else:

print("Такого имени нет")

case "все":
```

```
for name in sorted(phonebook):

print(f"{name}: {phonebook[name]}")

case "выход":

print("До свидания!")

break

case _:

print("Не понимаю команду")
```

```
Записная книжка. Команды: добавить, найти, удалить, все, выход.

> добавить

Имя: ветеринар

Телефон: +7 495 777-88-99

Записала: ветеринар

> найти

Кого ищем? бабушка

+7 900 111-22-33

> все

бабушка: +7 900 111-22-33

ветеринар: +7 495 777-88-99

> удалить

Кого удалить? сосед

Такого имени нет

> выход

До свидания!
```

Книжка работает, пока программа запущена. Закройте её, и все записи пропадут: словарь живёт в памяти компьютера. Как сохранять данные в файл и загружать их при следующем запуске, вы узнаете в главе 10.

Попробуйте сами 6.12. Добавьте в записную книжку команду "сколько", которая выводит количество записей, и сделайте так, чтобы команда "добавить" предупреждала, если такое имя уже есть, и спрашивала, заменить ли номер.

Итоги

Словарь хранит пары "ключ - значение" в фигурных скобках: {"молоко": 89}. Значение достают и меняют по ключу: prices["молоко"]; del удаляет пару, in проверяет наличие ключа.

Отсутствующий ключ даёт KeyError; метод get(ключ, по_умолчанию) возвращает значение по умолчанию или None.

Ключи - неизменяемые значения (строки, числа, кортежи), значения - любые. Словарь помнит порядок добавления пар.

Перебор: for key in d - ключи, d.values() - значения, d.items() - пары. max(d, key=d.get) находит ключ с наибольшим значением.

Вложенные структуры (список словарей, словарь списков, словарь словарей) описывают почти любые данные; доступ - цепочкой скобок.

Множество хранит уникальные значения без порядка: set(список) убирает повторы, операции | & - ^ дают объединение, пересечение и разности. Пустое множество - set(), а не {}.

frozendict (новинка Python 3.15) - неизменяемый словарь: подходит для констант, может быть ключом словаря и элементом множества; | создаёт новый frozendict с изменениями.

Глава 7. Строки и текст

Большая часть данных, с которыми работают программы, - это текст. Имена, адреса, сообщения, отзывы, списки покупок, номера телефонов, которые каждый записывает по-своему. В этой главе вы научитесь приводить текст в порядок: менять регистр, искать и заменять, разбивать на части и склеивать обратно. Узнаете, как красиво выводить числа и даты. А ещё разберёмся с особенностями русского языка, о которые спотыкаются даже опытные программисты: регистр, сортировка и буква ё.

Методы строк

В главе 5 мы познакомились с методами списков: append, sort и другими. У строк методов ещё больше. Вызываются они так же, через точку.

Листинг 7.1. string_methods.py

```
name = " барсик рыжий "


print(f"[{name}]")

print(f"[{name.strip()}]")

print(name.strip().upper())

print(name.strip().capitalize())

print(name.strip().title())

print("КОТ УЧЁНЫЙ".lower())

print("Барсик".startswith("Бар"), "Барсик".endswith("ик"))

print("2026".isdigit(), "20 26".isdigit(), "кот".isalpha())
```

```
[ барсик рыжий ]

[барсик рыжий]

БАРСИК РЫЖИЙ

Барсик рыжий

Барсик Рыжий

кот учёный

True True

True False True
```

Разберём по порядку:

strip() убирает пробелы (и переводы строк) по краям строки. Квадратные скобки в выводе нужны только для того, чтобы увидеть, где строка начинается и где заканчивается. Есть ещё lstrip() и rstrip(): они чистят только левый или только правый край.

upper() и lower() переводят всю строку в верхний или нижний регистр, то есть в большие или маленькие буквы.

capitalize() делает заглавной первую букву строки, а title() - первую букву каждого слова.

startswith() и endswith() проверяют, начинается ли строка с заданного куска и заканчивается ли им.

isdigit() проверяет, состоит ли строка только из цифр, а isalpha() - только из букв. В строке "20 26" есть пробел, поэтому это не только цифры.

Методы можно вызывать цепочкой: name.strip().upper() сначала убирает пробелы, а потом к результату применяет upper. Строки неизменяемы, поэтому методы никогда не меняют исходную строку, а возвращают новую. Если результат нужен дальше, его сохраняют в переменную: name = name.strip().

Новое в Python 3.15. Если вы пришли из JavaScript или Java и по привычке напишете toUpperCase, Python подскажет, как это называется у него:

```
>>> "привет".toUpperCase()

Traceback (most recent call last):

File "<python-input-0>", line 1, in <module>

"привет".toUpperCase()

^^^^^^^^^^^^^^^^^^^

AttributeError: 'str' object has no attribute 'toUpperCase'. Did you mean '.upper'?
```

Ответ "да" в любом виде

В главе 3 мы мучились с ответами "да" и "Да" и писали двойную проверку. Теперь можно навести порядок: убрать лишние пробелы, перевести ответ в нижний регистр и сравнивать один раз.

Листинг 7.2. answer_yes.py

```
answer = input("Хотите сосиску? ").strip().lower()


if answer in ["да", "д", "yes", "ага"]:

print("Держите сосиску!")

else:

print("Ну и ладно, Барсик съест.")
```

```
Хотите сосиску? ДА

Держите сосиску!
```

Человек нажал Caps Lock и случайно поставил пробелы, а программа всё равно поняла. Проверка answer in [...] сравнивает ответ сразу с несколькими вариантами. Такая "нормализация" ввода (приведение к единому виду) - первое, что делают с любым текстом от человека.

Попробуйте сами 7.1. Спросите у человека имя и выведите его так, чтобы первая буква была заглавной, а остальные строчными, даже если человек ввёл "мАША" или " маша ".

Попробуйте сами 7.2. Спросите год рождения и, прежде чем превращать его в число, проверьте методом isdigit, что введены только цифры. Если нет, выведите "Год нужно ввести цифрами".

Попробуйте сами 7.3. Дан список файлов ["отчёт.docx", "кот.jpg", "смета.xlsx", "дача.JPG", "договор.pdf"]. Выведите только картинки (файлы с расширением jpg в любом регистре).

Поиск и замена

Чтобы узнать, есть ли в строке какой-то кусок, достаточно оператора in, знакомого по спискам. А чтобы узнать, где именно он находится, есть метод find. Он возвращает индекс первого вхождения или -1, если ничего не найдено.

```
>>> text = "Барсик съел сосиску и сардельку"

>>> "сосиск" in text

True

>>> text.find("сосиск")

12

>>> text.find("колбас")

-1

>>> text.count("с")

6

>>> text.replace("сосиску", "кашу")

'Барсик съел кашу и сардельку'

>>> text

'Барсик съел сосиску и сардельку'
```

Метод count считает, сколько раз кусок встречается в строке, а replace заменяет все вхождения одного куска другим. Обратите внимание на последнюю строку: исходный текст не изменился. replace вернул новую строку, а мы её никуда не сохранили. Барсик по-прежнему съел сосиску.

Есть ещё метод index, похожий на find, но если кусок не найден, он вызывает ошибку ValueError, а не возвращает -1. Если отсутствие - нормальная ситуация, удобнее find или in.

Приводим телефоны к одному виду

Каждый записывает телефон по-своему: со скобками, с дефисами, через восьмёрку. Чтобы сравнивать номера и искать по ним, их приводят к единому виду. Оставим в номере только цифры, а восьмёрку в начале заменим на семёрку.

Листинг 7.3. phones.py

```
phones = ["+7 (900) 111-22-33", "8 900 111 22 33", "89001112233", "111-22-33"]


for phone in phones:
```

```
digits = ""

for char in phone:

if char.isdigit():

digits += char

if len(digits) == 11 and digits.startswith("8"):

digits = "7" + digits[1:]

if len(digits) == 11:

print(f"{phone:<20} -> +{digits}")

else:

print(f"{phone:<20} -> непонятный номер")
+7 (900) 111-22-33 -> +79001112233

8 900 111 22 33 -> +79001112233

89001112233 -> +79001112233

111-22-33 -> непонятный номер
```

Цикл по символам оставляет только цифры. Потом проверяем: если цифр 11 и первая - восьмёрка, меняем её на семёрку срезом digits[1:], который отрезает первый символ. Первые три записи оказались одним и тем же номером, а у последней не хватает кода, и программа честно в этом признаётся.

Попробуйте сами 7.4. Напишите программу, которая заменяет в тексте все "ё" на "е" и считает, сколько замен сделано. Проверьте на фразе "Ёжик пришёл в ёлочный лес, а Барсик всё ещё спит".

Попробуйте сами 7.5. Спросите у человека адрес электронной почты и проверьте самые простые правила: в адресе ровно один символ @, после него есть точка, и в адресе нет пробелов.

Разбиение и склейка

Метод split разбивает строку на части и возвращает список. Без аргументов он делит по пробелам (и по любому количеству пробелов подряд), а с аргументом - по указанному разделителю.

```
>>> "Барсик спит на батарее".split()

['Барсик', 'спит', 'на', 'батарее']

>>> "молоко,хлеб,сыр".split(",")

['молоко', 'хлеб', 'сыр']

>>> "12.03.2022".split(".")

['12', '03', '2022']

>>> ", ".join(["молоко", "хлеб", "сыр"])

'молоко, хлеб, сыр'

>>> "-".join("Барсик")

'Б-а-р-с-и-к'
```

Обратная операция - метод join ("соединить"). Он склеивает элементы списка в одну строку, вставляя между ними ту строку, у которой вызван. Запись выглядит непривычно: разделитель стоит впереди, ", ".join(список). Зато так можно склеить любые строки, даже буквы одного слова.

Разберём строку с покупками, которую кто-то прислал в сообщении:

Листинг 7.4. parse_purchases.py

```
message = "молоко 89; хлеб 45 ;сыр 450; корм для Барсика 320"


purchases = {}

for part in message.split(";"):

part = part.strip()

name, price = part.rsplit(" ", 1)

purchases[name] = int(price)


print(purchases)

print(f"Итого: {sum(purchases.values())} руб.")
{'молоко': 89, 'хлеб': 45, 'сыр': 450, 'корм для Барсика': 320}
```

Итого: 904 руб.

Сначала делим сообщение по точкам с запятой и получаем куски вроде " хлеб 45 ". Каждый кусок очищаем от пробелов по краям. Потом отделяем цену от названия. Просто split() тут не подойдёт: "корм для Барсика 320" развалится на четыре слова. Метод rsplit(" ", 1) делит строку по пробелу справа и не больше одного раза, поэтому получаем ровно два куска: название целиком и цену.

Ещё пример - инициалы из полного имени:

Листинг 7.5. initials.py

```
full_name = input("Фамилия Имя Отчество: ")

last, first, middle = full_name.split()

print(f"{last.capitalize()} {first[0].upper()}. {middle[0].upper()}.")
```

```
Фамилия Имя Отчество: петров сергей иванович

Петров С. И.
```

split() дал список из трёх слов, и он сразу распаковался в три переменные. Если слов окажется не три, распаковка вызовет ошибку ValueError. В главе 9 мы научимся перехватывать такие ошибки.

Для многострочного текста есть метод splitlines(): он делит текст на строки по переводам строк. Он ещё пригодится нам в проектах, например на сайте с рецептами в главе 27.

Попробуйте сами 7.6. Дана строка с оценками ученика "5, 4, 3, 5, 5, 4". Получите из неё список чисел и посчитайте средний балл.

Попробуйте сами 7.7. Спросите у человека фразу и выведите её слова в обратном порядке: "Барсик любит сосиски" должно превратиться в "сосиски любит Барсик".

Форматирование чисел

Мы уже знаем формат :.2f для двух знаков после точки. Возможностей у f-строк гораздо больше. Вот самые полезные:

```
>>> price = 1234567.891

>>> f"{price:.2f}"

'1234567.89'

>>> f"{price:,.2f}"

'1,234,567.89'

>>> f"{price:_.2f}"

'1_234_567.89'

>>> f"{0.256:.1%}"

'25.6%'

>>> f"{7:03}"

'007'

>>> f"{42:+}"

'+42'
```

:, и :_ разделяют разряды запятыми или подчёркиваниями;
:.1% умножает число на 100 и добавляет знак процента;
:03 дополняет число нулями слева до трёх знаков: удобно для номеров вроде "007";
:+ всегда показывает знак числа, даже у положительных.

В России разряды принято разделять пробелами, а дробную часть - запятой: 1 234 567,89. Такого формата в f-строках нет, но его легко получить заменами: сначала оформить по-английски, потом поменять запятые на пробелы, а точку - на запятую.

Листинг 7.6. rubles.py

```
amount = 1234567.891

english = f"{amount:,.2f}"

russian = english.replace(",", " ").replace(".", ",")

print(f"По-английски: {english}")

print(f"По-русски: {russian} руб.")
```

```
По-английски: 1,234,567.89
```

По-русски: 1 234 567,89 руб.

Выравнивание и таблицы

Формат может задать и ширину поля: сколько символов занимает значение. Знак < прижимает значение влево, > - вправо, ^ - к центру. Перед знаком можно указать символ-заполнитель.

Листинг 7.7. budget_table.py

```
expenses = [

("Продукты", 18500, 0.535),

("Коммуналка", 6800, 0.197),

("Развлечения", 4000, 0.116),

("Транспорт", 3200, 0.092),

("Кот", 2100, 0.061),

]


print(f"{' Расходы за ноябрь ':=^40}")

print(f"{'Категория':<15}{'Сумма':>12}{'Доля':>13}")

print("-" * 40)

for name, amount, share in expenses:

print(f"{name:<15}{amount:>12,}{share:>13.1%}")

print("=" * 40)
```

```
========== Расходы за ноябрь ==========

Категория Сумма Доля

----------------------------------------

Продукты 18,500 53.5%

Коммуналка 6,800 19.7%

Развлечения 4,000 11.6%

Транспорт 3,200 9.2%

Кот 2,100 6.1%
```

```
========================================
```

Заголовок {' Расходы за ноябрь ':=^40} занимает 40 символов, текст стоит по центру, а свободное место заполнено знаками =. Названия категорий прижаты влево в поле шириной 15, суммы и доли - вправо. Благодаря этому столбцы ровные, как в бухгалтерской ведомости. Обратите внимание, что форматы можно комбинировать: :>12, - это и ширина 12 с выравниванием вправо, и разделитель разрядов.

Попробуйте сами 7.8. Выведите таблицу умножения на 5 так, чтобы числа стояли ровным столбцом справа: 5 x 1 = 5 и так до 5 x 10 = 50.

Попробуйте сами 7.9. Пусть программа спрашивает сумму вклада и процент в год и выводит, сколько будет на вкладе через 1, 2, 3, 4 и 5 лет (проценты начисляются раз в год на всю сумму). Суммы выведите по-русски: с пробелами между разрядами и запятой перед копейками.

Даты

Для дат в Python есть модуль datetime. Как подключать модули, мы видели в главе 4 на примере random. Из модуля datetime нам понадобится тип date - дата.

Листинг 7.8. new_year.py

```
from datetime import date


today = date.today()

new_year = date(today.year + 1, 1, 1)

days_left = (new_year - today).days


print("Сегодня:", today)

print("По-нашему:", today.strftime("%d.%m.%Y"))

print(f"До Нового года осталось дней: {days_left}")
```

```
Сегодня: 2026-11-19

По-нашему: 19.11.2026

До Нового года осталось дней: 43
```

Запись from datetime import date значит "из модуля datetime взять date". После этого date.today() возвращает сегодняшнюю дату, а date(2027, 1, 1) создаёт дату из года, месяца и дня. У даты есть свойства year, month и day: например, today.year - текущий год. Даты можно вычитать: разность - это промежуток времени (тип timedelta), а его свойство days - количество дней.

Если вы запустите программу в другой день, цифры будут другими. Это нормально: программа считает от сегодняшней даты.

Метод strftime (string format time, "время в строку") превращает дату в текст по шаблону. В шаблоне %d - день, %m - месяц, %Y - год из четырёх цифр. Есть и обратная функция strptime (string parse time): она разбирает строку с датой по такому же шаблону. С ней можно спросить дату у человека.

Даты по-русски

У strftime есть шаблоны для названий месяцев и дней недели, но выдают они их по-английски:

```
>>> from datetime import date

>>> date(2026, 12, 31).strftime("%d %B %Y, %A")

'31 December 2026, Thursday'
```

Можно настроить русский язык через модуль locale, но такие настройки по-разному работают на разных компьютерах. Надёжнее сделать свои списки названий. Заодно сразу запишем месяцы в родительном падеже, как нужно в датах: "31 декабря", а не "31 декабрь".

Листинг 7.9. russian_date.py

```
from datetime import date, datetime


MONTHS = ["января", "февраля", "марта", "апреля", "мая", "июня", "июля",

"августа", "сентября", "октября", "ноября", "декабря"]

WEEKDAYS = ["понедельник", "вторник", "среда", "четверг",

"пятница", "суббота", "воскресенье"]


text = input("Дата рождения (ДД.ММ.ГГГГ): ")

birthday = datetime.strptime(text, "%d.%m.%Y").date()


month = MONTHS[birthday.month - 1]

weekday = WEEKDAYS[birthday.weekday()]

print(f"Вы родились {birthday.day} {month} {birthday.year} года, "

f"день недели: {weekday}.")


age_days = (date.today() - birthday).days

print(f"С тех пор прошло {age_days} дней.")
```

```
Дата рождения (ДД.ММ.ГГГГ): 12.03.2022

Вы родились 12 марта 2022 года, день недели: суббота.

С тех пор прошло 1713 дней.
```

Номер месяца идёт от 1 до 12, а индексы списка - от 0 до 11, поэтому MONTHS[birthday.month - 1]. Метод weekday() возвращает номер дня недели: 0 - понедельник, 6 - воскресенье, ровно как индексы в нашем списке. Функция datetime.strptime разбирает строку и возвращает дату со временем, а .date() оставляет только дату.

Длинная f-строка в print разбита на две: две строки подряд внутри скобок Python склеивает в одну. Это удобный способ уложиться в ширину экрана.

Попробуйте сами 7.10. Напишите программу, которая спрашивает дату в формате ДД.ММ.ГГГГ и сообщает, какой это будет день недели. Узнайте, на какой день недели выпадет ваш следующий день рождения.

Попробуйте сами 7.11. Помидоры созревают примерно через 110 дней после посева. Спросите дату посева и выведите ожидаемую дату первого урожая по-русски. Подсказка: к дате

можно прибавить промежуток timedelta(days=110), если подключить его из того же модуля: from datetime import datetime, timedelta.

Русский текст: регистр, сортировка и буква ё

Методы upper, lower, capitalize прекрасно работают с русскими буквами. А вот сортировка русских слов готовит сюрпризы. Посмотрите:

```
>>> words = ["ёжик", "Яблоко", "арбуз", "ель", "Ёлка", "банан", "яблоко"]

>>> sorted(words)

['Ёлка', 'Яблоко', 'арбуз', 'банан', 'ель', 'яблоко', 'ёжик']
```

Получилась каша. Python сортирует строки по номерам символов в таблице Юникод (Unicode) - всемирной таблице, где у каждого символа любого языка свой номер. В этой таблице заглавные русские буквы идут раньше строчных, поэтому "Яблоко" оказалось раньше "арбуза". А буква ё стоит отдельно от остального алфавита: заглавная Ё - перед всеми русскими буквами, а строчная ё - после всех. Так "Ёлка" попала в начало, а "ёжик" - в самый конец.

Регистр победить легко: сортировать по строке в нижнем регистре. Для этого в sorted передают key=str.lower, то есть "сравнивай не сами слова, а их варианты в нижнем регистре":

```
>>> words = ["ёжик", "Яблоко", "арбуз", "ель", "Ёлка", "банан", "яблоко"]

>>> sorted(words, key=str.lower)

['арбуз', 'банан', 'ель', 'Яблоко', 'яблоко', 'ёжик', 'Ёлка']
```

Стало лучше, но ёжик с ёлкой всё ещё в конце, после яблок. Чтобы ё встала на своё место после е, нужно сравнивать слова по нашему, русскому алфавиту. Для этого напишем маленькую функцию, которая превращает слово в список номеров букв в алфавите. Функции подробно разберём в следующей главе, а пока достаточно понять идею: def создаёт новую команду russian_key, а return говорит, что она возвращает.

Листинг 7.10. russian_sort.py

```
ALPHABET = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя"



def russian_key(word):

return [ALPHABET.find(char) for char in word.lower()]



words = ["ёжик", "Яблоко", "арбуз", "ель", "Ёлка", "банан", "яблоко", "енот"]

print(sorted(words, key=russian_key))
['арбуз', 'банан', 'ель', 'енот', 'ёжик', 'Ёлка', 'Яблоко', 'яблоко']
```

Теперь порядок как в бумажном словаре: "ель" и "енот" на е, за ними "ёжик" и "Ёлка" на ё, и яблоки в самом конце. Функция russian_key для каждой буквы слова ищет её номер в строке ALPHABET, и слова сравниваются по этим номерам. Символы, которых нет в алфавите (пробелы, дефисы, латинские буквы), получают номер -1 и при сравнении оказываются впереди русских букв.

Поиск, который не замечает ё

Буква ё доставляет хлопоты и при поиске. Кто-то пишет "ёлка", кто-то "елка", и обычный поиск считает эти слова разными. Решение - сравнивать нормализованные строки: в нижнем регистре и с заменой ё на е.

Листинг 7.11. search_yo.py

```
recipes = ["Ёлочные пряники", "Салат с ёжиками", "Котлеты по-домашнему",

"Пирог с черёмухой", "Морс из клюквы"]


query = input("Что ищем? ")

query_norm = query.lower().replace("ё", "е")


found = [r for r in recipes if query_norm in r.lower().replace("ё", "е")]

if found:

print("Нашлось:", ", ".join(found))

else:

print("Ничего не нашлось")
```

```
Что ищем? ЕЛОЧН

Нашлось: Ёлочные пряники
Что ищем? черемух

Нашлось: Пирог с черёмухой
```

Поисковый запрос и каждый рецепт приводятся к одному виду, и сравниваются уже они. При этом в результатах рецепты выводятся в исходном виде, с большими буквами и ё. Этот приём нам пригодится в проекте сайта с рецептами в главе 28.

Совет. Для сравнения строк без учёта регистра в Python есть метод casefold(). Для русского языка он работает так же, как lower(), но для некоторых других языков точнее: например, немецкую ß он превращает в ss. Если программа будет работать с текстами на разных языках, используйте casefold().

Попробуйте сами 7.12. Отсортируйте по русскому алфавиту список фамилий: ["Ёлкин", "Ефимов", "Яковлев", "абрамова", "Жуков", "Елизарова"]. Обратите внимание, что одна фамилия записана с маленькой буквы.

Попробуйте сами 7.13. Напишите программу, которая проверяет, является ли фраза палиндромом, не обращая внимания на пробелы, регистр и разницу между е и ё. Проверьте на фразе "А роза упала на лапу Азора".

Собираем всё вместе: разбор отзывов

Магазин кормов для животных получил отзывы покупателей. Посчитаем, сколько в отзывах слов и предложений и какие слова встречаются чаще всего. Короткие слова вроде "и", "в", "на" пропустим: они встречаются везде и ни о чём не говорят.

Листинг 7.12. reviews.py

```
text = """Корм отличный! Кот ест с удовольствием, просит ещё.

Доставка быстрая, корм свежий. Рекомендую.

Мой кот корм не ест. Кошка тоже. Верните деньги!

Ёжику корм тоже понравился, а коту нет."""


sentences = text.count(".") + text.count("!") + text.count("?")

words = text.lower().replace("ё", "е").split()

clean = [word.strip(".,!?") for word in words]


counts = {}

for word in clean:

if len(word) > 3:

counts[word] = counts.get(word, 0) + 1


print(f"Символов: {len(text)}, слов: {len(clean)}, предложений: {sentences}")

print("Чаще всего встречаются:")

for word in sorted(counts, key=counts.get, reverse=True)[:4]:

print(f" {word:<15}{counts[word]}")
```

```
Символов: 183, слов: 29, предложений: 8

Чаще всего встречаются:

корм 4

тоже 2

отличный 1
```

удовольствием 1

Здесь пригодилось многое из этой и прошлых глав: тройные кавычки для многострочного текста, count для подсчёта знаков конца предложения, lower и replace для нормализации, split для деления на слова, strip с аргументом, который убирает с краёв слов знаки препинания, и подсчёт словарём из главы 6. Главное слово отзывов предсказуемо: "корм". Барсик бы сказал, что это слово главное и в жизни.

Попробуйте сами 7.14. Дополните программу: пусть она считает, сколько отзывов (строк) в тексте содержат слово "кот" в любой форме ("кот", "кота", "коту") и сколько из них со словом "не". Подсказка: метод splitlines() делит текст на строки.

Итоги

Методы строк возвращают новую строку, исходная не меняется: strip() чистит края, lower() и upper() меняют регистр, capitalize() и title() делают заглавные буквы, isdigit() и isalpha() проверяют содержимое.

Ввод от человека нормализуют перед сравнением: input().strip().lower().

find ищет кусок строки (или возвращает -1), count считает, replace заменяет; in проверяет, есть ли кусок в строке.

split делит строку на список, rsplit(" ", 1) - один раз справа, join склеивает список в строку через разделитель.

Форматы f-строк: :, разряды, :.1% проценты, :03 нули слева, :<15, :>12, :^40 выравнивание; русский формат денег получают заменами.

Модуль datetime: date.today(), date(год, месяц, день), разность дат в днях, strftime и strptime с шаблонами %d.%m.%Y. Названия месяцев и дней по-русски надёжнее брать из своих списков.

Python сортирует строки по номерам символов Юникода: большие буквы раньше маленьких, ё вне алфавита. key=str.lower решает проблему регистра, своя функция с русским алфавитом - проблему ё; при поиске ё заменяют на е.

Глава 8. Функции

Мы уже пользовались множеством функций: print, input, len, sorted, round. Каждая из них - это готовая команда с именем, которая что-то делает. В этой главе вы научитесь создавать свои функции. Это важнейший шаг: функции позволяют разбить большую программу на понятные кусочки, не повторять один и тот же код и давать действиям человеческие имена.

Зачем нужны функции

Представьте, что бабушка записывает рецепты в тетрадь. В рецепте пирога написано "приготовить тесто" и дальше десять строк, как его готовить. В рецепте ватрушек - снова "приготовить тесто" и те же десять строк. И в рецепте пирожков. Гораздо удобнее один раз записать рецепт теста на отдельной странице, назвать его "Тесто дрожжевое" и в остальных рецептах писать просто: "приготовить тесто (см. стр. 12)".

Функция (function) - это и есть такой рецепт на отдельной странице: кусок кода с именем, который можно выполнить в любом месте программы, просто назвав его. Если в рецепте теста обнаружится ошибка, исправить её нужно будет в одном месте, а не в трёх.

У программистов есть принцип DRY (Don't Repeat Yourself, "не повторяйся"). Если вы скопировали кусок кода второй раз, это повод сделать из него функцию.

Объявление и вызов

Функцию объявляют (define) словом def, за которым идут имя, круглые скобки и двоеточие. Тело функции записывается с отступом.

Листинг 8.1. greet.py

```
def greet():

print("Привет!")

print("Барсик рад вас видеть.")



greet()

print("...прошло полчаса...")

greet()
```

```
Привет!

Барсик рад вас видеть.

...прошло полчаса...

Привет!

Барсик рад вас видеть.
```

Строки def greet(): и две строки с отступом создают функцию, но ничего не выводят: они только объясняют Python, что делать, когда функцию позовут. А зовут её вызовом (call): имя и круглые скобки, greet(). Каждый вызов выполняет тело функции заново.

Функцию нужно объявить до того, как вызывать: Python читает программу сверху вниз и о функции, объявленной ниже, ещё не знает. Имена функций пишут так же, как имена переменных: маленькими буквами через подчёркивание. Хорошее имя функции - глагол или глагол с дополнением: greet, print_menu, count_words, потому что функция что-то делает.

Совет. Между объявлениями функций и остальным кодом принято оставлять по две пустые строки. Python они не нужны, но так программу легче читать. Так требует PEP 8 - свод правил оформления кода на Python, о котором мы подробно поговорим в главе 14.

Параметры и аргументы

Функция, которая всегда делает одно и то же, полезна, но скучна. Интереснее, когда ей можно что-то передать. Для этого в скобках при объявлении пишут параметры (parameters) - имена для значений, которые функция получит.

Листинг 8.2. greet_name.py

```
def greet(name, greeting="Привет"):

print(f"{greeting}, {name}!")



greet("Маша")

greet("Сергей Иванович", "Здравствуйте")

greet(greeting="Доброе утро", name="Барсик")
```

```
Привет, Маша!

Здравствуйте, Сергей Иванович!

Доброе утро, Барсик!
```

Значения, которые передают при вызове, называются аргументами (arguments). При вызове greet("Маша") параметр name получает значение "Маша". У параметра greeting есть значение по умолчанию (default value) "Привет": если аргумент не передан, используется оно.

Аргументы можно передавать двумя способами. Позиционные (positional) передаются по порядку: первый аргумент - первому параметру, второй - второму. Именованные (keyword) передаются с именем параметра, greeting="Доброе утро", и тогда порядок не важен. Именованные аргументы делают вызов понятнее: greet(name="Барсик") читается без заглядывания в объявление функции.

Если забыть обязательный аргумент, Python скажет об этом прямо:

```
>>> def greet(name, greeting="Привет"):

... print(f"{greeting}, {name}!")

...

>>> greet()

Traceback (most recent call last):

File "<python-input-1>", line 1, in <module>

greet()

~~~~~^^

TypeError: greet() missing 1 required positional argument: 'name'
```

"Функции greet() не хватает одного обязательного позиционного аргумента: name".

Функции с параметрами удобно использовать для повторяющегося оформления. Например, для строк чека:

Листинг 8.3. receipt_lines.py

```
def print_line(name, price, quantity=1):

total = price * quantity

print(f"{name:<18}{quantity:>3} x {price:>6.2f} = {total:>8.2f}")



def print_separator(char="-", width=40):

print(char * width)



print_separator("=")

print_line("Молоко", 89.90, 2)

print_line("Хлеб", 45)

print_line("Корм для Барсика", 320, 3)

print_separator()
==========================================

Молоко 2 x 89.90 = 179.80

Хлеб 1 x 45.00 = 45.00

Корм для Барсика 3 x 320.00 = 960.00

----------------------------------------
```

Форматирование строки чека записано один раз, в функции print_line. Если завтра понадобится выводить чек по-другому, изменить придётся только её.

Частая ошибка. Не делайте значением по умолчанию изменяемый объект, например пустой список: def add_item(item, basket=[]). Этот список создаётся один раз, когда Python читает объявление функции, а не при каждом вызове. В результате все вызовы без аргумента basket будут складывать товары в одну и ту же корзину. Правильно писать basket=None, а внутри функции: if basket is None: basket = []. Проверка is None значит "это именно None": так это значение принято сравнивать.

Попробуйте сами 8.1. Напишите функцию print_box(text), которая выводит текст в рамке из звёздочек. Ширина рамки должна зависеть от длины текста.

Попробуйте сами 8.2. Добавьте функции print_box параметр char со значением по умолчанию "*", чтобы рамку можно было рисовать любым символом. Вызовите функцию с позиционным и с именованным аргументом.

Возвращаемое значение

Пока наши функции только выводили что-то на экран. Но чаще функция должна вычислить результат и отдать его программе, чтобы с ним можно было работать дальше. Для этого есть инструкция return ("вернуть").

Листинг 8.4. discount.py

```
def price_with_discount(price, percent):

return round(price * (100 - percent) / 100, 2)



milk = price_with_discount(89.90, 10)

cheese = price_with_discount(450, 15)

print(f"Молоко со скидкой: {milk:.2f} руб.")

print(f"Сыр со скидкой: {cheese:.2f} руб.")

print(f"Вместе: {milk + cheese:.2f} руб.")

print(f"Пять пачек корма: {price_with_discount(320, 5) * 5:.2f} руб.")
```

```
Молоко со скидкой: 80.91 руб.

Сыр со скидкой: 382.50 руб.

Вместе: 463.41 руб.

Пять пачек корма: 1520.00 руб.
```

return завершает функцию и отдаёт значение туда, откуда её вызвали. Вызов price_with_discount(89.90, 10) в программе как бы заменяется результатом, 80.91, и его можно сохранить в переменной, сложить с другим числом или подставить в f-строку. Именно так работают встроенные функции: len("Барсик") возвращает 6.

print или return

Новички часто путают вывод на экран и возврат значения. Посмотрите:

Листинг 8.5. print_vs_return.py

```
def area_print(width, height):

print(width * height)



def area_return(width, height):

return width * height
```

```
result1 = area_print(3, 4)

result2 = area_return(3, 4)

print("result1 =", result1)

print("result2 =", result2)
12

result1 = None

result2 = 12
```

Первая функция вывела 12 на экран, но ничего не вернула. Функция без return возвращает специальное значение None ("ничего"), с которым мы встречались в главе 6. Поэтому в result1 оказалось None, и посчитать с ним ничего нельзя. Правило простое: если результат функции нужен программе, функция должна его возвращать, а выводить его на экран - дело того, кто функцию вызвал.

Несколько значений и ранний выход

Функция может вернуть сразу несколько значений: на самом деле она возвращает кортеж, который удобно распаковать.

Листинг 8.6. harvest_stats.py

```
def stats(numbers):

if not numbers:

return 0, 0, 0

average = sum(numbers) / len(numbers)

return min(numbers), max(numbers), round(average, 2)



harvest = [2.5, 3.1, 1.8, 4.2, 3.6, 2.9, 0.7]

low, high, avg = stats(harvest)

print(f"Урожай за неделю: от {low} до {high} кг, в среднем {avg} кг")

print("Пустая неделя:", stats([]))
Урожай за неделю: от 0.7 до 4.2 кг, в среднем 2.69 кг

Пустая неделя: (0, 0, 0)
```

Обратите внимание на первые строки функции. Если список пустой, считать среднее нельзя (будет деление на ноль), поэтому функция сразу возвращает нули. После return функция заканчивается, и остальные строки не выполняются. Такой приём называют ранним выходом (early return): сначала разобраться с особыми случаями, а потом спокойно заниматься основной работой.

Функции, которые отвечают "да" или "нет", возвращают True или False. Их принято называть с is_ или has_: "является ли", "есть ли". Вспомним високосные годы из главы 3:

Листинг 8.7. leap.py

```
def is_leap(year):

return (year % 4 == 0 and year % 100 != 0) or year % 400 == 0



def days_in_year(year):

return 366 if is_leap(year) else 365



for year in [1900, 2000, 2024, 2026]:

print(f"{year}: високосный - {is_leap(year)}, дней {days_in_year(year)}")
```

```
1900: високосный - False, дней 365

2000: високосный - True, дней 366

2024: високосный - True, дней 366

2026: високосный - False, дней 365
```

Функция is_leap возвращает результат логического выражения: True или False. Писать if ...: return True else: return False не нужно, выражение уже и есть ответ. А функция days_in_year использует is_leap: функции могут вызывать друг друга, и из маленьких понятных функций складываются большие программы.

Попробуйте сами 8.3. Напишите функцию grade(points), которая по баллам от 0 до 100 возвращает оценку (как в задании 3.5), и выведите оценки для баллов 95, 80, 61 и 30.

Попробуйте сами 8.4. Напишите функцию cat_human_age(cat_age), которая возвращает возраст кота в человеческих годах по формуле из главы 3. Используйте ранний выход для котят.

Попробуйте сами 8.5. Напишите функцию split_bill(total, people, tip_percent=10), которая возвращает два значения: сумму чаевых и сумму с каждого человека (с чаевыми). Округлите обе до копеек.

Области видимости

Переменные, созданные внутри функции, живут только внутри неё. Их называют локальными (local). Снаружи их не видно:

Листинг 8.8. scope.py

```
discount = 10 # глобальная переменная



def apply_discount(price):

result = price * (100 - discount) / 100 # result - локальная

return result



print(apply_discount(500))

print(result)
450.0

Traceback (most recent call last):

File "C:\pyprojects\scope.py", line 10, in <module>

print(result)

^^^^^^

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

Функция спокойно прочитала переменную discount, объявленную снаружи. Такие переменные, созданные вне всех функций, называют глобальными (global). А вот переменная result существует, только пока работает функция, и после её завершения исчезает. Отсюда NameError в последней строке.

Область, где видна переменная, называется областью видимости (scope). Это полезное ограничение: можно называть переменные внутри функций как угодно и не бояться, что они случайно испортят переменные с такими же именами в другой части программы.

А что будет, если внутри функции попытаться изменить глобальную переменную?

Листинг 8.9. scope_error.py

```
visits = 0



def count_visit():

visits += 1
```

```
print(f"Заходов Барсика на кухню: {visits}")



count_visit()
Traceback (most recent call last):

File "C:\pyprojects\scope_error.py", line 9, in <module>

count_visit()

~~~~~~~~~~~^^

File "C:\pyprojects\scope_error.py", line 5, in count_visit

visits += 1

^^^^^^

UnboundLocalError: cannot access local variable 'visits' where it is not associated with a value
```

Присваивание внутри функции (а visits += 1 - это присваивание) делает переменную локальной. Python видит, что внутри функции есть локальная visits, пытается прочитать её значение для сложения, а значения ещё нет. "Нельзя обратиться к локальной переменной visits, у которой нет значения".

Python разрешает изменить глобальную переменную, если объявить её внутри функции словом global visits. Но опытные программисты делают так очень редко: когда функции тайком меняют общие переменные, программу трудно понять и трудно чинить. Лучше передавать значения через параметры и получать результат через return:

Листинг 8.10. scope_fixed.py

```
def count_visit(visits):

visits += 1

print(f"Заходов Барсика на кухню: {visits}")

return visits



visits = 0

visits = count_visit(visits)

visits = count_visit(visits)
Заходов Барсика на кухню: 1

Заходов Барсика на кухню: 2
```

Теперь всё, что функция получает, видно в её параметрах, а всё, что отдаёт, - в return. Такую функцию легко понять, не читая остальную программу.

Попробуйте сами 8.6. Не запуская программу, предскажите, что она выведет. Потом проверьте.

```
x = 1


def f():

x = 2

print("внутри:", x)


f()

print("снаружи:", x)
```

Произвольное число аргументов

Функции print можно передать сколько угодно аргументов. Свою функцию тоже можно так научить. Если перед именем параметра поставить звёздочку, он соберёт все лишние позиционные аргументы в кортеж. По традиции такой параметр называют args (arguments). А две звёздочки собирают лишние именованные аргументы в словарь, его по традиции называют kwargs (keyword arguments). Но можно дать им и говорящие имена, как в листинге ниже.

Листинг 8.11. pizza.py

```
def total(*prices):

return sum(prices)



def order_pizza(size, *toppings, **details):

print(f"Пицца {size} см")

if toppings:

print("Начинка:", ", ".join(toppings))

for key, value in details.items():

print(f" {key}: {value}")



print("Сумма:", total(89, 45, 450))

print("Сумма:", total())

order_pizza(30, "сыр", "грибы", "ветчина",

адрес="ул. Садовая, 5", оплата="картой")

order_pizza(25)
```

```
Сумма: 584

Сумма: 0

Пицца 30 см

Начинка: сыр, грибы, ветчина

адрес: ул. Садовая, 5

оплата: картой
```

Пицца 25 см

В функции total все аргументы попали в кортеж prices, а sum его сложила. Вызов без аргументов тоже работает: кортеж просто пустой. В order_pizza первый аргумент попал в size, остальные позиционные - в toppings, а именованные - в словарь details.

Звёздочки работают и в обратную сторону, при вызове. Если значения уже лежат в списке, звёздочка "выкладывает" их как отдельные аргументы:

```
>>> def total(*prices):

... return sum(prices)

...

>>> basket = [89, 45, 450]

>>> total(*basket)

584

>>> print(*basket, sep=" + ")

89 + 45 + 450
```

Параметр sep у print задаёт разделитель между значениями: по умолчанию это пробел.

Попробуйте сами 8.7. Напишите функцию average(*numbers), которая возвращает среднее всех переданных чисел, а если чисел нет - None.

Документация и аннотации типов

Хорошая функция объясняет сама себя. Для этого первой строкой в теле функции пишут строку документации (docstring) в тройных кавычках: что функция делает, что принимает и что возвращает. А после имён параметров можно указать аннотации типов (type hints): какого типа значения ожидаются.

Листинг 8.12. documented.py

```
def price_with_discount(price: float, percent: float = 0) -> float:

"""Возвращает цену со скидкой, округлённую до копеек.


price - цена в рублях, percent - скидка в процентах (от 0 до 100).

"""

return round(price * (100 - percent) / 100, 2)



help(price_with_discount)

print(price_with_discount(450, 15))
```

```
Help on function price_with_discount in module __main__:


price_with_discount(price: float, percent: float = 0) -> float

Возвращает цену со скидкой, округлённую до копеек.


price - цена в рублях, percent - скидка в процентах (от 0 до 100).


382.5
```

Функция help показывает объявление функции и её документацию. В VS Code то же самое появляется во всплывающей подсказке, когда вы наводите мышь на имя функции или начинаете набирать её вызов.

Аннотации читаются так: price: float - "параметр price, дробное число", -> float - "функция возвращает дробное число". Для списков и словарей пишут, что лежит внутри: list[str] - список строк, dict[str, int] - словарь со строковыми ключами и целыми значениями. Если значение может отсутствовать, пишут str | None - "строка или None".

Важно понимать: сам Python аннотации не проверяет. Если передать в price_with_discount строку, он не остановит вас заранее, а упадёт уже внутри функции при вычислении. Аннотации - подсказки для человека и для редактора: VS Code может подчеркнуть вызов, где тип явно не подходит, ещё до запуска: для этого в настройках (Ctrl+,) найдите Type Checking Mode и выберите basic. В книге мы будем писать аннотации в функциях проектов, где они особенно помогают.

Попробуйте сами 8.8. Добавьте строки документации и аннотации типов к функциям grade и cat_human_age из заданий 8.3 и 8.4. Посмотрите на них с помощью help.

Короткие функции: lambda

Иногда функция нужна на один раз и настолько короткая, что давать ей имя и писать def неудобно. Для таких случаев есть лямбда-функции (lambda functions): lambda x: x * 2 - это функция, которая принимает x и возвращает x * 2. Слово return не пишется, результатом становится выражение после двоеточия.

Чаще всего лямбды используют как параметр key в sorted, min и max. В главе 5 мы обещали отсортировать прайс-лист по цене. Выполняем обещание:

Листинг 8.13. sort_lambda.py

```
price_list = [

("молоко", 89),

("сыр", 450),

("хлеб", 45),

("корм для Барсика", 320),

]


by_price = sorted(price_list, key=lambda item: item[1])

print("От дешёвого к дорогому:", by_price)


priciest = max(price_list, key=lambda item: item[1])

print("Самое дорогое:", priciest[0])


pets = {"Барсик": 5.6, "Мурка": 3.9, "Пушок": 4.4}

for name, weight in sorted(pets.items(), key=lambda pair: pair[1]):

print(f"{name}: {weight} кг")
```

От дешёвого к дорогому: [('хлеб', 45), ('молоко', 89), ('корм для Барсика', 320), ('сыр', 450)]

Самое дорогое: сыр

Мурка: 3.9 кг

Пушок: 4.4 кг

Барсик: 5.6 кг

Для каждого элемента списка sorted вызывает лямбду и сравнивает элементы по тому, что она вернула. lambda item: item[1] берёт из кортежа второй элемент, цену. Получается "сортировать товары по цене".

Лямбда может содержать только одно выражение: никаких циклов, if с блоками и нескольких строк. Если функция получается сложной или нужна в нескольких местах, напишите обычную функцию через def и дайте ей понятное имя.

Попробуйте сами 8.9. Отсортируйте список слов ["кот", "Абрикос", "ёжик", "банан", "Дыня"] без учёта регистра с помощью лямбды. Затем отсортируйте его по последней букве слова.

Попробуйте сами 8.10. Дан список учеников с оценками: [("Аня", 4.8), ("Боря", 3.9), ("Вика", 4.5)]. С помощью max и min с лямбдой найдите лучшего и худшего ученика.

Собираем всё вместе: викторина на функциях

В главе 3 мы написали викторину с повторяющимся кодом, в главе 5 убрали повторы с помощью списка. Теперь разложим её на функции, и каждая будет отвечать за одно дело.

Листинг 8.14. quiz_functions.py

```
QUESTIONS = [

("Сколько часов в сутки спит кошка: 5, 10 или 15?", "15"),

("Чувствуют ли кошки сладкий вкус? (да/нет)", "нет"),

("Видит ли кошка в полной темноте? (да/нет)", "нет"),

("Как называется детёныш кошки?", "котёнок"),

]



def normalize(text: str) -> str:

"""Приводит ответ к единому виду: без пробелов по краям, строчными."""

return text.strip().lower().replace("ё", "е")



def ask(number: int, question: str, right_answer: str) -> bool:

"""Задаёт вопрос и возвращает True, если ответ верный."""

answer = input(f"{number}. {question} ")

if normalize(answer) == normalize(right_answer):

print("Верно!")

return True

print(f"Нет, правильный ответ: {right_answer}.")

return False



def verdict(score: int, total: int) -> str:

"""Возвращает оценку по доле правильных ответов."""
```

```
share = score / total
if share == 1:
return "Вы настоящий знаток кошек!"
if share >= 0.5:
return "Неплохо! Барсик вами доволен."
return "Барсик смотрит с укором. Попробуйте ещё раз."

def run_quiz(questions: list[tuple[str, str]]) -> int:
"""Проводит викторину и возвращает число правильных ответов."""
score = 0
for number, (question, right_answer) in enumerate(questions, start=1):
if ask(number, question, right_answer):
score += 1
return score

def main():
print("Викторина: хорошо ли вы знаете кошек?")
score = run_quiz(QUESTIONS)
print(f"\nПравильных ответов: {score} из {len(QUESTIONS)}.")
print(verdict(score, len(QUESTIONS)))

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

Викторина: хорошо ли вы знаете кошек?

1. Сколько часов в сутки спит кошка: 5, 10 или 15? 15

```
Верно!

2. Чувствуют ли кошки сладкий вкус? (да/нет) Нет

Верно!

3. Видит ли кошка в полной темноте? (да/нет) да

Нет, правильный ответ: нет.

4. Как называется детёныш кошки? Котенок

Верно!


Правильных ответов: 3 из 4.

Неплохо! Барсик вами доволен.
```

Программа стала длиннее, но каждую её часть можно понять отдельно. normalize приводит ответы к единому виду, поэтому "Нет" и "нет", "Котенок" и "котёнок" засчитываются одинаково. ask задаёт один вопрос. verdict выбирает оценку. run_quiz проводит викторину целиком, а main (главная) собирает всё вместе.

Последние две строки - распространённый в Python обычай. Переменная __name__ равна "__main__", когда файл запущен как программа. Если же этот файл подключат к другой программе как модуль (об этом - в главе 11), main() сам не запустится, и другая программа сможет пользоваться, например, функцией normalize, не запуская викторину. В программах этой книги, где есть функция main, мы будем всегда заканчивать файл такими строками.

Попробуйте сами 8.11. Добавьте в викторину функцию ask_name(), которая спрашивает имя игрока и возвращает его, и пусть итоговая оценка выводится с обращением по имени.

Попробуйте сами 8.12. Напишите функцию print_progress(score, total), которая рисует полоску прогресса из символов, например [####------] 4/10, и вызывайте её после каждого вопроса.

Итоги

Функция - именованный кусок кода. Её объявляют словом def с параметрами в скобках и вызывают по имени со скобками и аргументами.

Параметры могут иметь значения по умолчанию; аргументы передают по порядку (позиционные) или по имени (именованные). Изменяемые значения по умолчанию ([]) - ловушка, вместо них пишут None.

return завершает функцию и возвращает значение; функция без return возвращает None. Вывод на экран и возврат значения - разные вещи.

Функция может вернуть несколько значений кортежем; ранний выход через return упрощает обработку особых случаев.

Переменные внутри функции локальные и снаружи не видны. Менять глобальные переменные из функции можно через global, но лучше передавать данные параметрами и возвращать результат.

*args собирает лишние позиционные аргументы в кортеж, **kwargs - именованные в словарь; звёздочка при вызове раскладывает список на аргументы.

Строка документации и аннотации типов (price: float, -> float) объясняют функцию человеку и редактору; Python их не проверяет.

lambda - короткая безымянная функция из одного выражения, чаще всего для key в sorted, min и max. Обычай if __name__ == "__main__": main() запускает программу, только когда файл запущен напрямую.

Глава 9. Ошибки и отладка

Мы уже не раз видели, как программа падает с сообщением об ошибке. Пора разобраться с ошибками всерьёз. В этой главе вы научитесь читать длинные трассировки, перехватывать ошибки, чтобы программа не падала от первого неверного ввода, создавать свои ошибки, искать причины проблем в отладчике и вести журнал работы программы.

Три вида ошибок

Ошибки в программах бывают трёх видов.

Синтаксические ошибки (syntax errors) - нарушение правил записи: пропущенная скобка, лишний отступ. Их Python находит до запуска, и программа не начинает работать вовсе. Мы встречались с ними в главе 1.

Ошибки времени выполнения (runtime errors) случаются во время работы: деление на ноль, обращение к несуществующему ключу словаря, превращение в число строки "двадцать пять". В Python их называют исключениями (exceptions), потому что они прерывают обычный ход программы. Программа выполняется до строки с ошибкой и падает.

Логические ошибки (logic errors) - самые коварные. Программа работает и ничего не сообщает, но результат неправильный. Помните скидку 2% вместо 10% из главы 3? Python не видит таких ошибок: он делает ровно то, что написано, а написано не то. Их находят тестами (глава 13) и отладкой.

Как читать трассировку

Когда ошибка случается глубоко внутри функций, трассировка становится длиннее. Возьмём программу, которая делит расходы семьи на её членов.

Листинг 9.1. family_share.py

```
def share(total, people):

return total / people



def print_shares(expenses, family):

for category, amount in expenses.items():

print(f"{category}: по {share(amount, len(family)):.2f} руб.")



def main():

expenses = {"продукты": 18500, "коммуналка": 6800}

family = []

print_shares(expenses, family)



main()
Traceback (most recent call last):

File "C:\pyprojects\family_share.py", line 16, in <module>

main()

~~~~^^

File "C:\pyprojects\family_share.py", line 13, in main

print_shares(expenses, family)

~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^

File "C:\pyprojects\family_share.py", line 7, in print_shares

print(f"{category}: по {share(amount, len(family)):.2f} руб.")
```

```
~~~~~^^^^^^^^^^^^^^^^^^^^
File "C:\pyprojects\family_share.py", line 2, in share
return total / people
~~~~~~^~~~~~~~
ZeroDivisionError: division by zero
```

Трассировка похожа на стопку записок "кто кого позвал". Читаем, как договорились в главе 1, снизу вверх.

1. Последняя строка: что случилось. ZeroDivisionError: division by zero - деление на ноль.
2. Над ней - где именно: файл, строка 2, функция share, строка return total / people. Значки под строкой показывают, что виновато именно деление.
3. Выше - кто позвал share: строка 7, функция print_shares.
4. Ещё выше - кто позвал print_shares: строка 13, функция main.
5. И в самом верху - вызов main() в строке 16, в основном тексте программы.

Ошибка случилась в share, но причина не там. Функция делит честно, просто ей передали ноль людей. Поднимаемся выше по трассировке и находим настоящую причину: в main список family пустой. Это частая история: место, где программа упала, и место, где ошибка совершена, - разные места. Трассировка помогает пройти по цепочке вызовов от одного к другому.

Совет. В больших программах трассировка проходит через файлы библиотек, которые писали не вы. Ищите в ней строки со своими файлами: ошибка почти наверняка в вашем коде, а не в библиотеке, которой пользуются миллионы людей.

Частые исключения

Вот исключения, которые встречаются чаще всего. Половину из них мы уже видели.

Исключение	**Когда возникает**	**Пример**
ValueError	значение подходящего типа, но неправильное	int("двадцать")
TypeError	операция с неподходящим типом	"5" + 1
NameError	неизвестное имя	опечатка pirnt
IndexError	индекс за пределами списка	[1, 2][5]

KeyError
нет такого ключа в словаре
prices["колбаса"]

AttributeError
у значения нет такого метода
[1].push(2)

ZeroDivisionError
деление на ноль
1 / 0

FileNotFoundError
нет такого файла
открыть несуществующий файл

Имя исключения - уже половина диагноза. Привыкайте по имени угадывать, где искать причину.

Попробуйте сами 9.1. Напишите пять строк, каждая из которых вызывает своё исключение из таблицы. Запустите каждую в интерактивном режиме и прочитайте сообщение.

Попробуйте сами 9.2. Исправьте программу из листинга 9.1 так, чтобы она не падала, если в семье никого нет, а выводила "Делить расходы не на кого". Подумайте, где лучше сделать проверку: в share, в print_shares или в main.

Перехват исключений: try и except

Программа не должна падать оттого, что человек ошибся при вводе. Для этого исключения перехватывают (catch) конструкцией try - except: "попробуй сделать это, а если случится такое-то исключение, сделай вот что".

Листинг 9.2. ask_age.py

```
while True:

text = input("Сколько вам лет? ")

try:

age = int(text)

break

except ValueError:

print(f"'{text}' - это не число. Введите возраст цифрами.")


print(f"Через год вам будет {age + 1}")
```

```
Сколько вам лет? двадцать пять

'двадцать пять' - это не число. Введите возраст цифрами.

Сколько вам лет? 25 лет

'25 лет' - это не число. Введите возраст цифрами.

Сколько вам лет? 25

Через год вам будет 26
```

В блоке try записан код, который может упасть. Если всё прошло гладко, блок except пропускается. Если в блоке try случилось исключение ValueError, Python не падает, а прерывает блок try (до break дело не доходит) и выполняет блок except. Цикл повторяется, пока человек не введёт настоящее число.

Обратите внимание, что после except указан тип исключения. Перехватывается только ValueError, а все остальные ошибки по-прежнему роняют программу. Это правильно: перехватывать нужно только те ошибки, которые вы ожидаете и знаете, как обработать.

Несколько видов исключений

Если в блоке try может случиться разное, для каждого случая пишут свой except:

Листинг 9.3. divide.py

```
try:

total = float(input("Сумма счёта: "))
```

```
people = int(input("Сколько человек? "))

print(f"С каждого: {total / people:.2f} руб.")

except ValueError:

print("Нужно ввести число.")

except ZeroDivisionError:

print("Делить на ноль человек не получится.")
Сумма счёта: 3500

Сколько человек? 0

Делить на ноль человек не получится.
Сумма счёта: много

Нужно ввести число.
```

Python проверяет блоки except сверху вниз и выполняет первый подходящий. Если одинаково обрабатываются несколько исключений, их перечисляют в скобках: except (ValueError, ZeroDivisionError):. Начиная с Python 3.14 скобки можно не ставить: except ValueError, ZeroDivisionError:. В книге мы будем ставить скобки: так понятнее и работает в любой версии.

Само исключение можно получить в переменную, чтобы показать его сообщение:

```
>>> try:

... int("двадцать")

... except ValueError as error:

... print("Ошибка:", error)

...

Ошибка: invalid literal for int() with base 10: 'двадцать'
```

Частая ошибка. Не пишите except: без типа исключения и не перехватывайте все ошибки подряд через except Exception: без веской причины. Такой блок проглотит и опечатку в имени переменной, и ошибку в логике, и программа будет молча работать неправильно. Ловите только то, что ожидаете.

Попробуйте сами 9.3. Напишите функцию ask_int(prompt), которая спрашивает целое число, пока человек не введёт его правильно, и возвращает это число.

Попробуйте сами 9.4. Напишите программу, которая спрашивает название товара и выводит его цену из словаря prices. Если товара нет, перехватите KeyError и выведите "Такого товара нет". Потом перепишите программу без try, с методом get. Какой вариант вам понятнее?

Блоки else и finally

У конструкции try есть ещё два необязательных блока. Блок else выполняется, если исключения не было. Блок finally выполняется всегда: и после успеха, и после ошибки.

Листинг 9.4. try_else_finally.py

```
def check_weight(text):

try:

weight = float(text)

except ValueError:

print("Ошибка: вес должен быть числом.")

else:

if weight > 6:

print(f"{weight} кг - Барсику пора на диету.")

else:

print(f"{weight} кг - Барсик в форме.")

finally:

print("Взвешивание окончено.")



check_weight("6.4")

check_weight("пять")
```

```
6.4 кг - Барсику пора на диету.

Взвешивание окончено.

Ошибка: вес должен быть числом.

Взвешивание окончено.
```

Зачем else, если можно написать эти строки прямо в try? Чтобы в try осталось только то, что действительно может бросить ожидаемое исключение. Тогда except ValueError не перехватит случайно ошибку из другой строки.

Блок finally нужен для уборки за собой: закрыть файл, разорвать соединение с сетью, вывести итог. Он выполнится, даже если в try случится исключение, которое никто не перехватил, и программа после этого упадёт.

Попробуйте сами 9.5. Перепишите программу деления счёта из листинга 9.3 так, чтобы расчёт с каждого человека выводился в блоке else, а в блоке finally выводилось "Спасибо, что пользуетесь калькулятором!".

Свои исключения

Исключения можно не только перехватывать, но и создавать. Инструкция raise ("поднять") вызывает исключение. Это нужно, когда функция получила данные, с которыми не может работать, и должна сообщить об этом тому, кто её вызвал.

Листинг 9.5. piggy_bank.py

```
class NotEnoughMoneyError(Exception):

"""В копилке не хватает денег."""



def withdraw(balance, amount):

if amount <= 0:

raise ValueError("сумма должна быть больше нуля")

if amount > balance:

raise NotEnoughMoneyError(f"в копилке {balance} руб., нужно {amount}")

return balance - amount



balance = 2500

for amount in [700, -50, 3000, 1800]:

try:

balance = withdraw(balance, amount)

print(f"Взяли {amount} руб., осталось {balance} руб.")

except ValueError as error:

print("Ошибка:", error)

except NotEnoughMoneyError as error:

print("Не хватает денег:", error)
```

```
Взяли 700 руб., осталось 1800 руб.

Ошибка: сумма должна быть больше нуля

Не хватает денег: в копилке 1800 руб., нужно 3000
```

Взяли 1800 руб., осталось 0 руб.

Первые две строки создают своё исключение NotEnoughMoneyError. Слово class объявляет новый тип, о классах мы подробно поговорим в главе 12. Пока достаточно запомнить запись: своё исключение - это класс, у которого в скобках указано Exception, а внутри можно написать только строку документации. Имена исключений по традиции заканчиваются на Error.

Функция withdraw не выводит сообщений и не решает, что делать с ошибкой. Её дело - сообщить: "так нельзя, и вот почему". А решает вызывающая сторона: в нашей программе она выводит сообщение и переходит к следующей сумме. В другой программе та же функция могла бы использоваться иначе, например в Telegram-боте, который отвечает пользователю. Такое разделение делает функции полезными в разных программах.

Попробуйте сами 9.6. Напишите функцию set_age(age), которая вызывает ValueError, если возраст отрицательный или больше 150, и возвращает возраст, если всё в порядке. Проверьте её на значениях 25, -3 и 200, перехватывая исключения.

Попробуйте сами 9.7. Создайте исключение FridgeEmptyError и функцию take_from_fridge(fridge, product), которая удаляет продукт из списка fridge и вызывает FridgeEmptyError, если продукта там нет.

Отладка

Отладка (debugging) - поиск и исправление ошибок. Слово bug по-английски значит "жучок": по легенде, в 1947 году в огромной вычислительной машине нашли мотылька, застрявшего между контактами, и вклеили его в журнал с пометкой "первый настоящий баг". С тех пор отладка так и называется - "вылавливание жучков".

Отладка выводом

Самый простой способ - добавить print с переменными в подозрительные места, как мы делали в главе 4. Давайте найдём ошибку в программе, которая считает средний балл.

Листинг 9.6. average_bug.py

```
def average(numbers):

total = 0

for i in range(1, len(numbers)):

total += numbers[i]

return total / len(numbers)



grades = [5, 4, 3]

print(f"Средний балл: {average(grades):.2f}")
```

```
Средний балл: 2.33
```

Средний балл оценок 5, 4 и 3 должен быть 4, а программа выводит 2.33. Программа не упала, это логическая ошибка. Если добавить внутрь цикла строку print(f"{i=} {numbers[i]=} {total=}"), сразу будет видно, что цикл начинается с i=1 и пятёрка в сумму не попадает. Но есть способ удобнее.

Отладчик

Отладчик (debugger) - инструмент, который позволяет остановить программу на любой строке и заглянуть внутрь: посмотреть значения переменных, выполнить программу по одной строке, зайти внутрь функции.

В VS Code отладчик встроен и ставится вместе с расширением Python. Работать с ним так:

1. Щёлкните слева от номера строки, где хотите остановиться. Появится красная точка - точка останова (breakpoint).
2. Нажмите F5 (или "Выполнить → Начать отладку") и выберите "Python Debugger: Current File".
3. Программа остановится на красной точке. Слева в панели "Переменные" видны все переменные и их значения.

4. Дальше управляйте выполнением кнопками на панели вверху или клавишами: F10 - выполнить строку целиком (шаг с обходом), F11 - зайти внутрь вызываемой функции (шаг с заходом), Shift+F11 - выйти из функции, F5 - продолжить до следующей точки останова.

Поставьте точку останова на строку total += numbers[i] в листинге 9.6 и запустите отладку. Программа остановится, и в панели переменных будет i = 1. Значит, нулевой элемент пропущен, ошибка в range(1, len(numbers)).

Отладчик pdb

Если VS Code под рукой нет, в Python есть встроенный отладчик pdb (Python debugger). Чтобы им воспользоваться, вставьте в программу вызов breakpoint() - программа остановится на этом месте, и можно будет вводить команды отладчика.

Листинг 9.7. average_pdb.py

```
def average(numbers):

total = 0

for i in range(1, len(numbers)):

total += numbers[i]

return total / len(numbers)



grades = [5, 4, 3]

breakpoint()

print(f"Средний балл: {average(grades):.2f}")
> C:\pyprojects\average_pdb.py(9)<module>()

-> breakpoint()

(Pdb) p grades

[5, 4, 3]

(Pdb) n

> C:\pyprojects\average_pdb.py(10)<module>()

-> print(f"Средний балл: {average(grades):.2f}")

(Pdb) s

--Call--

> C:\pyprojects\average_pdb.py(1)average()
```

-> def average(numbers):

(Pdb) n

> C:\pyprojects\average_pdb.py(2)average()

-> total = 0

(Pdb) n

> C:\pyprojects\average_pdb.py(3)average()

-> for i in range(1, len(numbers)):

(Pdb) n

> C:\pyprojects\average_pdb.py(4)average()

-> total += numbers[i]

(Pdb) p i, numbers[i], total

(1, 4, 0)

(Pdb) c

Средний балл: 2.33

Строки после (Pdb) вводит человек, остальное отвечает отладчик. Самые нужные команды:

p выражение - напечатать значение (print): p grades, p i, total;

n - выполнить текущую строку и перейти к следующей (next);

s - зайти внутрь функции, которая вызывается в текущей строке (step);

l - показать код вокруг текущей строки (list);

c - продолжить выполнение до конца или до следующей остановки (continue);

q - прекратить отладку и выйти (quit).

Строка со стрелкой -> - та, которую отладчик выполнит следующей. После s мы оказались внутри функции average, прошли несколько строк и посмотрели на переменные: i равно 1, складывается четвёрка, а пятёрка пропущена. Ошибка найдена: нужно range(len(numbers)) или, ещё проще, for number in numbers: total += number. А лучше всего - sum(numbers). Когда ошибка исправлена, breakpoint() из программы удаляют.

Попробуйте сами 9.8. Исправьте ошибку в функции average и проверьте результат в отладчике VS Code: поставьте точку останова на строку с return и посмотрите значение total перед возвратом.

Логирование

Отладчик хорош, когда программа перед вами. А если программа работает сама по себе: бот круглые сутки отвечает клиентам, программа-наблюдатель каждый час проверяет цены? Что-то пошло не так ночью, а утром никто не помнит, что случилось. Для этого программы ведут журнал - лог (log). Записывать в него сообщения помогает модуль logging.

Листинг 9.8. logging_demo.py

```
import logging


logging.basicConfig(

level=logging.INFO,

format="%(asctime)s %(levelname)s %(message)s",

datefmt="%d.%m.%Y %H:%M:%S",

)


logging.debug("Подробности для отладки: этого не будет видно")

logging.info("Программа запущена")

logging.warning("Корм для Барсика заканчивается: осталось 2 пачки")


try:

portions = 10 / 0

except ZeroDivisionError:

logging.exception("Не удалось рассчитать порции")


logging.info("Программа завершена")
```

```
17.11.2026 12:03:19 INFO Программа запущена

17.11.2026 12:03:19 WARNING Корм для Барсика заканчивается: осталось 2 пачки

17.11.2026 12:03:19 ERROR Не удалось рассчитать порции

Traceback (most recent call last):

File "C:\pyprojects\logging_demo.py", line 14, in <module>
```

```
    portions = 10 / 0

    ~~~^~~

ZeroDivisionError: division by zero

17.11.2026 12:03:19 INFO Программа завершена
```

У каждого сообщения есть уровень важности. По возрастанию: DEBUG (подробности для отладки), INFO (обычные события), WARNING (предупреждение), ERROR (ошибка), CRITICAL (всё пропало). Параметр level=logging.INFO в basicConfig говорит: "записывать сообщения от INFO и важнее". Поэтому сообщение уровня DEBUG не появилось. Когда понадобится искать ошибку, достаточно поменять уровень на logging.DEBUG, и подробности появятся, а в обычной работе они не мешают.

Параметр format задаёт вид строки: время, уровень, сообщение. Время берётся текущее, у вас оно будет другим. Функция logging.exception пишет сообщение уровня ERROR и вместе с ним всю трассировку исключения. Вызывают её внутри блока except: программа не упала, а в журнале осталось всё, что нужно для поиска причины.

Журнал обычно пишут не на экран, а в файл. Для этого в basicConfig добавляют два параметра: filename="app.log" и encoding="utf-8". Тогда сообщения будут копиться в файле app.log рядом с программой, и утром можно будет прочитать, что происходило ночью. В проектах журнал нам ещё встретится: его ведут Telegram-бот (главы 25-26) и мини-CRM (глава 32), а программе-наблюдателю мы посоветуем журнал в файле (глава 24).

Совет. Чем logging лучше print? Сообщения журнала можно включать и выключать по уровням, не удаляя их из кода; в них автоматически пишется время; их можно направить в файл. Для временных проверок при отладке годится и print, а для всего, что должно остаться в программе, используйте logging.

Попробуйте сами 9.9. Добавьте журнал в программу из листинга 9.2: записывайте уровнем INFO каждый правильный ответ и уровнем WARNING каждую неудачную попытку ввода. Сделайте так, чтобы журнал писался в файл age.log.

Собираем всё вместе: надёжный расчёт поездки

В главе 2 мы написали расчёт стоимости поездки на дачу, и он падал от любого неверного ответа. Сделаем его надёжным: каждое число будем спрашивать функцией, которая не сдаётся, пока не получит правильный ответ, и проверяет, что число положительное.

Листинг 9.9. dacha_trip_safe.py

```
def ask_number(prompt, cast=float):

"""Спрашивает положительное число, пока не получит правильный ответ."""

while True:

text = input(prompt).strip().replace(",", ".")

try:

value = cast(text)

except ValueError:

print(" Это не число. Попробуйте ещё раз.")

continue

if value <= 0:

print(" Число должно быть больше нуля.")

continue

return value


def main():

distance = ask_number("Расстояние до дачи в одну сторону, км: ")

consumption = ask_number("Расход бензина, литров на 100 км: ")

fuel_price = ask_number("Цена литра бензина, руб.: ")

people = ask_number("Сколько человек едет? ", cast=int)


cost = distance * 2 * consumption / 100 * fuel_price

print(f"\nПоездка обойдётся в {cost:.2f} руб., "
```

```
    f"с каждого по {cost / people:.2f} руб.")


if __name__ == "__main__":

main()
Расстояние до дачи в одну сторону, км: 68 км

Это не число. Попробуйте ещё раз.

Расстояние до дачи в одну сторону, км: 68

Расход бензина, литров на 100 км: 8,5

Цена литра бензина, руб.: 62.4

Сколько человек едет? 0

Число должно быть больше нуля.

Сколько человек едет? три

Это не число. Попробуйте ещё раз.

Сколько человек едет? 3


Поездка обойдётся в 721.34 руб., с каждого по 240.45 руб.
```

Функция ask_number собрала в себе всё, что нужно для надёжного ввода. Запятая в дробных числах заменяется точкой, поэтому "8,5" тоже понимается правильно. Параметр cast - это функция, которой превращать текст в число: по умолчанию float, а для количества людей передаём int. Да, функцию можно передать в другую функцию как обычное значение, как мы передавали len и str.lower в параметр key.

Ноль людей теперь не пройдёт, и деления на ноль не будет. Программа больше не падает от неверного ввода, а спокойно объясняет, что не так.

Попробуйте сами 9.10. Добавьте в функцию ask_number параметр max_value со значением по умолчанию None. Если он задан, функция должна отклонять числа больше него. Используйте его, чтобы расход бензина нельзя было ввести больше 30 литров на 100 км.

Итоги

Ошибки бывают синтаксические (до запуска), исключения во время работы и логические (программа работает, но неправильно).

Трассировку читают снизу вверх: последняя строка - что случилось, строки выше - цепочка вызовов. Место падения и место настоящей ошибки часто разные.

try - except ТипОшибки перехватывает ожидаемые исключения; as error даёт доступ к сообщению. Не перехватывайте всё подряд.

else выполняется, если исключения не было, finally - всегда.

raise вызывает исключение; своё исключение - это класс, у которого в скобках указано Exception. Функция сообщает об ошибке, а решает, что с ней делать, вызывающий код.

Отладчик VS Code (точки останова, F10, F11) и встроенный pdb (breakpoint(), команды p, n, s, c) позволяют остановить программу и посмотреть на переменные.

Модуль logging ведёт журнал с уровнями DEBUG, INFO, WARNING, ERROR, CRITICAL; logging.exception записывает трассировку; журнал можно писать в файл.

Глава 10. Файлы

Все наши программы до сих пор страдали забывчивостью. Записная книжка из главы 6 теряла все номера, стоило её закрыть. Список покупок исчезал вместе с программой. Чтобы данные пережили выключение компьютера, их сохраняют в файлы. В этой главе вы научитесь читать и записывать текстовые файлы, разберётесь с кодировками (и узнаете, почему Python 3.15 сделал жизнь проще), научитесь работать с путями к файлам и папкам, а также с двумя самыми популярными форматами данных - CSV и JSON.

Запись и чтение

Работа с файлом похожа на работу с тетрадью: открыть, записать или прочитать, закрыть. Открывает файл функция open.

Листинг 10.1. diary_write.py

```
with open("diary.txt", "w", encoding="utf-8") as file:

file.write("12 мая. Посадили помидоры, 24 куста.\n")

file.write("13 мая. Барсик выкопал два куста. Посадили заново.\n")

file.write("15 мая. Укрыли рассаду: обещают заморозки.\n")


print("Записи сохранены в diary.txt")
```

```
Записи сохранены в diary.txt
```

Функции open передают три вещи: имя файла, режим (mode) и кодировку. Режим "w" (write) значит "открыть для записи": если файла нет, он будет создан, а если есть - очищен. Кодировку мы обсудим чуть ниже, пока просто запомните: всегда пишите encoding="utf-8".

Конструкция with ... as file: открывает файл и кладёт его в переменную file, а когда блок с отступом заканчивается, сама закрывает файл. Закрывать файлы важно: пока файл не закрыт, записанное может остаться в памяти и не попасть на диск. С with об этом можно не думать: файл закроется, даже если внутри блока случится ошибка.

Метод write записывает строку. В отличие от print, он не добавляет перевод строки сам, поэтому в конце каждой записи стоит \n.

Файл diary.txt появится в папке, откуда запущена программа. Его можно открыть в Блокноте или в VS Code. А можно прочитать программой:

Листинг 10.2. diary_read.py

```
with open("diary.txt", encoding="utf-8") as file:

text = file.read()

print(text)


print("Построчно:")

with open("diary.txt", encoding="utf-8") as file:

for number, line in enumerate(file, start=1):

print(number, line.rstrip("\n"))
```

```
12 мая. Посадили помидоры, 24 куста.

13 мая. Барсик выкопал два куста. Посадили заново.

15 мая. Укрыли рассаду: обещают заморозки.
```

Построчно:

1 12 мая. Посадили помидоры, 24 куста.

2 13 мая. Барсик выкопал два куста. Посадили заново.

3 15 мая. Укрыли рассаду: обещают заморозки.

Без указания режима файл открывается для чтения (режим "r", read). Метод read() читает весь файл в одну строку. А если перебирать файл циклом for, он отдаёт строки по одной. Это удобно для больших файлов: в памяти не нужно держать весь текст сразу. У каждой прочитанной строки в конце остаётся символ перевода строки, поэтому мы срезаем его методом rstrip("\n"). Иначе print добавил бы свой перевод строки, и между записями появились бы пустые строки.

Дописать в конец

Режим "w" стирает старое содержимое. Чтобы дописать новые строки в конец файла, нужен режим "a" (append, "добавить").

Листинг 10.3. diary_add.py

```
from datetime import date


entry = input("Что сегодня было на даче? ")

with open("diary.txt", "a", encoding="utf-8") as file:

file.write(f"{date.today().strftime('%d.%m.%Y')}. {entry}\n")


with open("diary.txt", encoding="utf-8") as file:

lines = file.readlines()

print(f"В дневнике записей: {len(lines)}. Последняя:")

print(lines[-1].rstrip("\n"))
```

```
Что сегодня было на даче? Собрали первые огурцы!

В дневнике записей: 4. Последняя:

19.11.2026. Собрали первые огурцы!
```

Метод readlines() читает файл в список строк, по элементу на строку. Каждый запуск программы добавляет в дневник новую запись, а старые остаются на месте.

А что, если файла нет? Попробуем прочитать несуществующий:

```
>>> open("nofile.txt", encoding="utf-8")

Traceback (most recent call last):
```

```
  File "<python-input-0>", line 1, in <module>

    open("nofile.txt", encoding="utf-8")

    ~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
```

FileNotFoundError - "нет такого файла или каталога". Его можно перехватить через try - except, как в главе 9, или заранее проверить, есть ли файл. Как проверять, расскажем в разделе о путях.

Попробуйте сами 10.1. Напишите программу, которая спрашивает у человека три дела на завтра и записывает их в файл todo.txt, каждое на новой строке, с номером.

Попробуйте сами 10.2. Напишите программу, которая читает diary.txt и выводит только записи, где упоминается Барсик.

Попробуйте сами 10.3. Измените программу из листинга 10.2 так, чтобы при отсутствии файла она не падала, а выводила "Дневник пока пуст".

Кодировки и UTF-8 по умолчанию

Компьютер хранит в файлах не буквы, а числа - байты (bytes). Кодировка (encoding) - это договорённость, каким числам какие буквы соответствуют. Если записать текст в одной кодировке, а прочитать в другой, получится бессмыслица. Её даже называют особым словом - кракозябры (по-английски mojibake). Посмотрим, как это выглядит:

```
>>> data = "Привет".encode("utf-8")

>>> data

b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

>>> data.decode("utf-8")

'Привет'

>>> data.decode("cp1251")

'РџСЂРёРІРµС,'
```

Метод encode превращает строку в байты по выбранной кодировке, a decode - обратно. В кодировке UTF-8 каждая русская буква занимает два байта, и "Привет" превращается в двенадцать байтов. Если прочитать их как Windows-1251 (cp1251) - старую кодировку для кириллицы в Windows, - получится "РџСЂёРІРµС,". Наверняка вы видели такое в письмах или на сайтах: это именно перепутанные кодировки.

Сегодня почти весь мир договорился использовать UTF-8: она умеет хранить буквы всех языков, математические знаки и даже смайлики. Но в Windows долго по умолчанию использовалась cp1251, и старые версии Python открывали файлы именно в ней, если кодировку не указать. Файл, записанный такой программой, в другой программе или на другом компьютере открывался кракозябрами.

Новое в Python 3.15. Начиная с этой версии Python всегда использует UTF-8 по умолчанию, если кодировка не указана, на любой системе. Это одно из самых полезных изменений для новичков на Windows. И всё же документация Python советует указывать кодировку явно: encoding="utf-8". Тогда программа будет одинаково работать и в Python 3.15, и в более старых версиях, которые могут оказаться у ваших друзей. В книге мы всегда пишем кодировку явно.

Явная кодировка нужна и тогда, когда файл пришёл из другой программы. Многие российские банки, бухгалтерские программы и старые сайты до сих пор выгружают файлы в cp1251. Если открыть такой файл как UTF-8, Python сообщит об ошибке:

Листинг 10.4. encodings.py

```
with open("bank.txt", "w", encoding="cp1251") as file:

file.write("Списание: Пятёрочка, 1250 руб.\n")


try:

with open("bank.txt", encoding="utf-8") as file:

print(file.read())
```

```
except UnicodeDecodeError as error:

print("Ошибка:", error)


with open("bank.txt", encoding="cp1251") as file:

print("В правильной кодировке:", file.read().rstrip())
```

```
Ошибка: 'utf-8' codec can't decode byte 0xd1 in position 0: invalid continuation byte

В правильной кодировке: Списание: Пятёрочка, 1250 руб.
```

UnicodeDecodeError - ошибка расшифровки: "кодек utf-8 не может расшифровать байт". Увидели такое сообщение при чтении файла - попробуйте encoding="cp1251". В главе 22 мы будем импортировать банковскую выписку, и там это пригодится.

Попробуйте сами 10.4. Превратите фразу "Барсик - мой кот" в байты UTF-8 методом encode, а потом прочитайте их как cp1251 с помощью decode в интерактивном режиме. Как выглядят кракозябры именно этой фразы?

Пути к файлам: модуль pathlib

До сих пор мы указывали только имя файла, и он создавался в текущей папке. Чтобы работать с файлами в других папках, создавать папки, перебирать файлы, нужен модуль pathlib. В нём есть тип Path - путь (path) к файлу или папке.

Листинг 10.5. paths.py

```
from pathlib import Path


folder = Path.cwd()

print("Текущая папка:", folder)


report = folder / "отчёты" / "ноябрь.txt"

print("Путь к отчёту:", report)

print("Имя файла:", report.name)

print("Без расширения:", report.stem)

print("Расширение:", report.suffix)

print("Папка отчёта:", report.parent.name)


report.parent.mkdir(exist_ok=True)

report.write_text("Потрачено 34 600 руб.\n", encoding="utf-8")

print("Файл существует?", report.exists())

print("Содержимое:", report.read_text(encoding="utf-8").rstrip())
```

```
Текущая папка: C:\pyprojects

Путь к отчёту: C:\pyprojects\отчёты\ноябрь.txt

Имя файла: ноябрь.txt

Без расширения: ноябрь

Расширение: .txt

Папка отчёта: отчёты

Файл существует? True
```

Содержимое: Потрачено 34 600 руб.

Path.cwd() (current working directory) возвращает текущую папку - ту, из которой запущена программа. Пути склеиваются знаком /: folder / "отчёты" / "ноябрь.txt". Это не деление, а "внутри": "в папке folder, в папке отчёты, файл ноябрь.txt". Знак / работает и на Windows: при выводе Python сам поставит обратные косые черты, как принято в этой системе. Так что проблема с \n в путях, о которой мы говорили в главе 2, решается сама собой.

У пути есть свойства: name - имя с расширением, stem - имя без расширения, suffix - расширение, parent - папка, в которой лежит файл. И методы:

exists() - существует ли; is_file() и is_dir() - файл это или папка;

mkdir() - создать папку; с exist_ok=True не ругается, если папка уже есть, а с parents=True создаёт и все промежуточные папки;

read_text() и write_text() - прочитать или записать весь файл одной командой, без with и open;

iterdir() - перебрать содержимое папки, glob("*.txt") - найти файлы по шаблону;

rename() - переименовать или переместить, unlink() - удалить файл.

Частая ошибка. Программа прекрасно находит файл data.txt, когда её запускают кнопкой в VS Code, и не находит, когда запускают из терминала. Причина в том, что относительный путь "data.txt" ищется в текущей папке, а она зависит от того, откуда запущена программа. Надёжный способ - строить путь от папки, где лежит сама программа: Path(__file__).parent / "data.txt". Переменная __file__ содержит путь к файлу программы.

Наводим порядок в загрузках

Папка "Загрузки" у многих выглядит как кладовка, куда всё сваливают годами. Напишем программу, которая раскладывает файлы по папкам в зависимости от расширения. Чтобы не трогать ваши настоящие загрузки, программа сначала создаст учебную папку с файлами-пустышками.

Листинг 10.6. sort_downloads.py

```
from pathlib import Path


FOLDERS = {

".jpg": "Картинки",

".png": "Картинки",

".docx": "Документы",

".pdf": "Документы",

".xlsx": "Таблицы",

".csv": "Таблицы",

}


downloads = Path("Загрузки_пример")
```

```
downloads.mkdir(exist_ok=True)

for name in ["кот.jpg", "договор.pdf", "смета.xlsx", "скриншот.PNG",

"рецепт.docx", "песня.mp3", "выписка.csv"]:

(downloads / name).touch()


for file in sorted(downloads.iterdir()):

if not file.is_file():

continue

folder_name = FOLDERS.get(file.suffix.lower(), "Прочее")

target_folder = downloads / folder_name

target_folder.mkdir(exist_ok=True)

file.rename(target_folder / file.name)

print(f"{file.name:<15} -> {folder_name}")


for folder in sorted(downloads.iterdir()):

names = sorted(f.name for f in folder.iterdir())

print(f"{folder.name}: {', '.join(names)}")
```

```
выписка.csv -> Таблицы

договор.pdf -> Документы

кот.jpg -> Картинки

песня.mp3 -> Прочее

рецепт.docx -> Документы

скриншот.PNG -> Картинки

смета.xlsx -> Таблицы

Документы: договор.pdf, рецепт.docx

Картинки: кот.jpg, скриншот.PNG
```

Прочее: песня.mp3

Таблицы: выписка.csv, смета.xlsx

Метод touch() создаёт пустой файл (или обновляет время изменения, если файл уже есть). Расширение мы переводим в нижний регистр, чтобы "скриншот.PNG" тоже попал в картинки. Для неизвестных расширений get возвращает папку "Прочее". А rename перемещает файл в новую папку.

В последнем цикле внутри sorted(...) стоит выражение без квадратных скобок: f.name for f in folder.iterdir(). Это выражение-генератор (generator expression), похожее на генератор списка, только оно не создаёт список целиком, а отдаёт элементы по одному. Когда такое выражение - единственный аргумент функции, квадратные скобки можно не писать.

Чтобы навести порядок в настоящей папке, достаточно заменить Path("Загрузки_пример") на Path.home() / "Downloads" и убрать создание пустышек. Но сначала проверьте программу на учебной папке: перемещение файлов - дело серьёзное, и ошибку в программе потом придётся исправлять руками.

Попробуйте сами 10.5. Напишите программу, которая выводит все файлы с расширением .py из текущей папки и их размер в байтах. Подсказка: размер файла даёт file.stat().st_size.

Попробуйте сами 10.6. Измените листинг 10.6 так, чтобы перед перемещением программа проверяла, нет ли в целевой папке файла с таким же именем, и если есть - пропускала файл с сообщением.

CSV: таблицы в тексте

CSV (comma-separated values, "значения, разделённые запятыми") - простейший формат таблиц. Каждая строка файла - строка таблицы, а значения в ней разделены запятыми. CSV умеют открывать и сохранять Excel, LibreOffice, Google Таблицы, его выгружают банки и интернет-магазины.

Есть тонкость, важная именно для России. В русской версии Excel запятая занята: она отделяет дробную часть числа. Поэтому русский Excel использует как разделитель точку с запятой. Вот файл с расходами семьи в таком формате:

дата;категория;сумма;кто

01.11.2026;продукты;2350;мама

02.11.2026;транспорт;600;папа

03.11.2026;кот;890;Дима

05.11.2026;продукты;4120;папа

07.11.2026;развлечения;1500;Дима

08.11.2026;продукты;1890;мама

10.11.2026;коммуналка;6800;папа

12.11.2026;кот;450;мама

Для работы с CSV в Python есть модуль csv. Удобнее всего читать файл через DictReader: он берёт названия столбцов из первой строки и отдаёт каждую следующую строку как словарь.

Листинг 10.7. expenses_report.py

```
import csv


totals = {}

with open("expenses.csv", encoding="utf-8", newline="") as file:

for row in csv.DictReader(file, delimiter=";"):

category = row["категория"]

totals[category] = totals.get(category, 0) + int(row["сумма"])


for category, amount in sorted(totals.items(), key=lambda p: -p[1]):

print(f"{category:<12}{amount:>6} руб.")
```

```
with open("summary.csv", "w", encoding="utf-8-sig", newline="") as file:

writer = csv.writer(file, delimiter=";")

writer.writerow(["категория", "сумма"])

for category, amount in totals.items():

writer.writerow([category, amount])

print("Итоги сохранены в summary.csv")
```

```
продукты 8360 руб.

коммуналка 6800 руб.

развлечения 1500 руб.

кот 1340 руб.

транспорт 600 руб.

Итоги сохранены в summary.csv
```

Обратите внимание на три детали.

Первая: delimiter=";" - разделитель столбцов. Для файлов из русского Excel - точка с запятой, для "международных" CSV - запятая (она стоит по умолчанию).

Вторая: newline="" при открытии файла. Модуль csv сам следит за переводами строк, и этот параметр не даёт Python вмешиваться. Без него на Windows в записанном файле появятся пустые строки между строками таблицы.

Третья: при записи указана кодировка utf-8-sig. Это тот же UTF-8, но с невидимой меткой в начале файла. Excel на Windows по этой метке понимает, что файл в UTF-8. Без неё он откроет русский текст кракозябрами. Поэтому правило такое: CSV для Excel записывайте в utf-8-sig.

Значения из CSV всегда приходят строками, даже числа. Поэтому сумму мы превращаем в число функцией int. А сортировка по -p[1], то есть по сумме с минусом, - короткий способ отсортировать по убыванию.

Попробуйте сами 10.7. Посчитайте по файлу expenses.csv, сколько потратил каждый член семьи.

Попробуйте сами 10.8. Запишите в CSV-файл таблицу умножения от 1 до 9 (первая строка - заголовки от 1 до 9) и откройте его в Excel или LibreOffice.

JSON: данные со структурой

CSV хорош для простых таблиц. Но как сохранить записную книжку, где у каждого человека несколько телефонов, или словарь словарей с питомцами? Для данных со сложной структурой есть формат JSON (JavaScript Object Notation). Выглядит он почти как словари и списки Python:

```
>>> import json

>>> pets = {"Барсик": {"возраст": 4, "прививки": ["бешенство", "чумка"]}}

>>> text = json.dumps(pets, ensure_ascii=False, indent=2)

>>> print(text)

{

"Барсик": {

"возраст": 4,

"прививки": [

"бешенство",

"чумка"

]

}

}

>>> json.loads(text)["Барсик"]["возраст"]

4
```

Функция json.dumps (dump string) превращает данные Python в строку JSON, а json.loads (load string) - обратно. Параметр ensure_ascii=False сохраняет русские буквы как есть: без него вместо "Барсик" в файле будут коды вида \u0411\u0430.... А indent=2 делает отступы, чтобы файл было удобно читать глазами.

В JSON можно сохранить словари, списки, строки, числа, True, False и None. Кортежи превращаются в списки, а множества JSON не поддерживает. Тип frozendict из Python 3.15 сохраняется как обычный словарь.

Для работы с файлами есть функции json.dump и json.load (без буквы s на конце). Теперь можно наконец научить записную книжку из главы 6 помнить номера.

Листинг 10.8. phonebook_json.py

```
import json

from pathlib import Path
```

```
BOOK_FILE = Path("phonebook.json")


def load_book():
if not BOOK_FILE.exists():
return {}
with open(BOOK_FILE, encoding="utf-8") as file:
return json.load(file)


def save_book(book):
with open(BOOK_FILE, "w", encoding="utf-8") as file:
json.dump(book, file, ensure_ascii=False, indent=2)


def main():
book = load_book()
print(f"Загружено записей: {len(book)}")
while True:
command = input("Команда (добавить, все, выход): ")
if command == "добавить":
name = input("Имя: ")
book[name] = input("Телефон: ")
save_book(book)
print("Сохранено.")
elif command == "все":
for name in sorted(book):
```

```
        print(f" {name}: {book[name]}")

        elif command == "выход":

        break



if __name__ == "__main__":

main()
```

```
Первый запуск: книжка пустая, добавим номер.
Загружено записей: 0

Команда (добавить, все, выход): добавить

Имя: бабушка

Телефон: +7 900 111-22-33

Сохранено.

Команда (добавить, все, выход): выход
Второй запуск, на следующий день:
Загружено записей: 1

Команда (добавить, все, выход): добавить

Имя: ветеринар

Телефон: +7 495 777-88-99

Сохранено.

Команда (добавить, все, выход): все

бабушка: +7 900 111-22-33

ветеринар: +7 495 777-88-99

Команда (добавить, все, выход): выход
```

Бабушкин номер никуда не делся. А вот как выглядит сам файл phonebook.json - его можно открыть в любом текстовом редакторе:

```
{

"бабушка": "+7 900 111-22-33",

"ветеринар": "+7 495 777-88-99"
```

```
}
```

Загрузка и сохранение вынесены в отдельные функции. Если книжка когда-нибудь переедет из JSON в базу данных (о базах данных - в главе 15), поменять придётся только их.

Совет. Сохраняйте данные сразу после изменения, а не только при выходе из программы. Если программа упадёт или компьютер выключится, последние изменения не пропадут.

Попробуйте сами 10.9. Добавьте в программу из листинга 10.8 команду "удалить". Не забудьте сохранить книжку после удаления.

Попробуйте сами 10.10. Сохраните в JSON-файл список покупок (список словарей с полями "товар", "количество", "куплено"), потом прочитайте его в другой программе и выведите только то, что ещё не куплено.

Что выбрать: TXT, CSV или JSON

Обычный текстовый файл - для текста, который читает человек: дневник, заметки, журнал работы программы.

CSV - для таблиц, особенно если их будут открывать в Excel или они пришли из банка и интернет-магазина.

JSON - для данных со сложной структурой: вложенные словари и списки, настройки программ. В JSON обмениваются данными сайты и приложения, мы встретимся с ним в главе 16.

Когда данных становится много и нужно искать в них по разным условиям, файлов уже не хватает. Тогда приходит время баз данных, и в главе 15 мы ими займёмся.

Итоги

Файл открывают в блоке with open(путь, режим, encoding="utf-8") as file: - тогда он закроется сам. Режимы: "r" - чтение (по умолчанию), "w" - запись с очисткой, "a" - дописать в конец.

read() читает весь файл, цикл for - построчно, readlines() - в список строк; write() записывает строку без автоматического перевода строки.

Кодировка - соответствие букв и байтов. В Python 3.15 по умолчанию UTF-8 на всех системах, но кодировку всё равно пишут явно; файлы от банков и старых программ часто в cp1251.

pathlib.Path - работа с путями: склейка через /, свойства name, stem, suffix, parent, методы exists, mkdir, iterdir, glob, read_text, write_text, rename. Путь к файлу рядом с программой: Path(__file__).parent / "имя".

CSV читают и пишут модулем csv с newline=""; русский Excel использует разделитель ;, а файлы для него пишут в кодировке utf-8-sig.

JSON хранит вложенные словари и списки: json.dump и json.load с ensure_ascii=False и indent=2.

Глава 11. Модули и пакеты

Программы растут. Викторина из главы 8 заняла пятьдесят строк, а проекты в этой книге будут занимать сотни. Держать всё в одном файле неудобно: трудно найти нужное место, нельзя взять удачную функцию в другую программу. Поэтому код раскладывают по модулям. В этой главе вы научитесь создавать свои модули и пакеты, познакомитесь с полезными модулями стандартной библиотеки, узнаете, как ставить чужие библиотеки из интернета и зачем каждому проекту своё виртуальное окружение.

Модули и import

Модуль (module) - это просто файл с кодом на Python. Любой файл .py можно подключить к другой программе командой import и пользоваться его функциями и переменными. Мы уже так делали с модулями random, datetime, json и csv, которые входят в Python. Теперь напишем свой.

Листинг 11.1. dacha_utils.py

```
"""Полезные функции для дачника."""


GRAMS_PER_SQ_M = 30 # сколько граммов удобрения на квадратный метр



def bed_area(length: float, width: float) -> float:

"""Площадь грядки в квадратных метрах."""

return round(length * width, 2)



def fertilizer_grams(area: float) -> int:

"""Сколько граммов удобрения нужно на площадь."""

return round(area * GRAMS_PER_SQ_M)



if __name__ == "__main__":

print("Проверка модуля:", bed_area(4, 1.2), fertilizer_grams(4.8))
```

Рядом, в той же папке, создадим программу, которая этим модулем пользуется:

Листинг 11.2. plan_beds.py

```
import dacha_utils

from dacha_utils import bed_area as area


beds = {"помидоры": (4, 1.2), "огурцы": (3, 1), "клубника": (5.5, 0.8)}


total = 0

for plant, (length, width) in beds.items():

size = area(length, width)
```

```
total += size

print(f"{plant:<10} {size:>5} кв. м")


grams = dacha_utils.fertilizer_grams(total)

print(f"Всего {total} кв. м, удобрения нужно {grams} г")

print(f"(норма {dacha_utils.GRAMS_PER_SQ_M} г на кв. м)")
```

```
помидоры 4.8 кв. м

огурцы 3 кв. м

клубника 4.4 кв. м

Всего 12.2 кв. м, удобрения нужно 366 г

(норма 30 г на кв. м)
```

Здесь показаны два способа подключения.

import dacha_utils подключает модуль целиком. К его содержимому обращаются через точку: dacha_utils.fertilizer_grams(total). Сразу видно, откуда взялась функция.

from dacha_utils import bed_area as area берёт из модуля одну функцию и даёт ей короткое имя area. Слово as ("как") позволяет переименовать то, что подключаем. Без него было бы просто from dacha_utils import bed_area. Через запятую можно подключить несколько имён: from datetime import date, timedelta.

Python ищет модули сначала в папке с запущенной программой, потом в стандартной библиотеке и среди установленных библиотек. Поэтому dacha_utils.py должен лежать рядом с plan_beds.py.

Зачем if __name__ == "__main__"

В конце модуля стоит знакомая по главе 8 конструкция. Когда файл запускают как программу, переменная __name__ равна "__main__", и проверочная строка выполняется:

```
Проверка модуля: 4.8 144
```

А когда файл подключают командой import, __name__ равна имени модуля, "dacha_utils", и проверка не выполняется. Поэтому при запуске plan_beds.py строки "Проверка модуля" в выводе не было. Так один файл может быть и модулем для других программ, и самостоятельной программой.

Частая ошибка. Не называйте свои файлы так же, как модули стандартной библиотеки: random.py, math.py, json.py, csv.py. Python найдёт ваш файл раньше настоящего модуля и подключит его.

Вот что бывает, если в папке с игрой лежит свой файл random.py:

Листинг 11.3. oops/random.py

```
print("Это мой файл random.py, а не настоящий модуль!")
```

Листинг 11.4. oops/game.py

```
import random
```

```
print(random.randint(1, 100))
Это мой файл random.py, а не настоящий модуль!

Traceback (most recent call last):

File "C:\pyprojects\oops\game.py", line 3, in <module>

print(random.randint(1, 100))

^^^^^^^^^^^^^^

AttributeError: module 'random' has no attribute 'randint' (consider renaming 'C:\pyprojects
\oops\random.py' since it has the same name as the standard library module named 'random' and
prevents importing that standard library module)
```

Python подключил не тот random, и функции randint в нём не оказалось. Хорошо, что современный Python в таком случае сам подсказывает: "подумайте о переименовании файла random.py, потому что у него то же имя, что у модуля стандартной библиотеки".

Попробуйте сами 11.1. Создайте модуль cat_tools.py с функцией cat_human_age из главы 8 и константой CAT_NAME = "Барсик". Подключите модуль в другой программе двумя способами: целиком и через from ... import.

Попробуйте сами 11.2. Добавьте в cat_tools.py проверочный блок if __name__ == "__main__":, который выводит возраст Барсика по-человечески. Убедитесь, что при подключении модуля он не срабатывает.

Пакеты

Когда модулей становится много, их складывают в папки. Папка с модулями называется пакетом (package). Чтобы Python понял, что папка - пакет, в неё кладут файл __init__.py. Он может быть пустым, а может содержать код, который выполнится при подключении пакета.

Листинг 11.5. garden/__init__.py

```
"""Пакет для планирования огорода."""
```

Листинг 11.6. garden/sowing.py

```
from datetime import date, timedelta


DAYS_TO_HARVEST = {"редис": 25, "огурцы": 50, "помидоры": 110}



def harvest_date(plant: str, sown: date) -> date:

return sown + timedelta(days=DAYS_TO_HARVEST[plant])
```

Листинг 11.7. garden/watering.py

```
def liters_per_day(area: float, hot: bool) -> float:

"""5 литров на квадратный метр, в жару вдвое больше."""

return area * (10 if hot else 5)
```

Листинг 11.8. use_garden.py

```
from datetime import date


from garden import watering

from garden.sowing import harvest_date


sown = date(2026, 5, 10)

for plant in ["редис", "огурцы", "помидоры"]:

ready = harvest_date(plant, sown)

print(f"{plant}: урожай около {ready.strftime('%d.%m.%Y')}")


print(f"Полив 4.8 кв. м в жару: {watering.liters_per_day(4.8, hot=True)} л")
```

```
редис: урожай около 04.06.2026

огурцы: урожай около 29.06.2026

помидоры: урожай около 28.08.2026
```

Полив 4.8 кв. м в жару: 48.0 л
Структура папок получилась такая:
pyprojects/

use_garden.py

garden/

__init__.py

sowing.py

watering.py

Модуль внутри пакета подключают через точку: from garden.sowing import harvest_date - "из модуля sowing пакета garden взять функцию harvest_date". Или from garden import watering - "из пакета garden взять модуль watering".

Обратите внимание на порядок import в начале программы. По соглашению сначала подключают модули стандартной библиотеки, потом сторонние библиотеки, потом свои модули, и между группами оставляют пустую строку. Python всё равно, а людям так легче понять, откуда что берётся.

Стандартная библиотека

Вместе с Python устанавливается стандартная библиотека (standard library) - сотни модулей на все случаи жизни. Про Python говорят, что он поставляется "с батарейками": многое можно сделать, ничего не устанавливая. Познакомимся с самыми полезными модулями. С некоторыми мы уже встречались, теперь узнаем о них больше.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.