

Переменные и Типы данных

Переменная — это просто именованная «коробка» в памяти компьютера, куда мы можем положить любое значение. В Python не нужно заранее говорить, что именно будет лежать в коробке, он поймет это сам.

Создаем переменные (знак "=" — это операция присваивания)

```
age = 25 # Целое число (integer)
price = 199.99 # Число с плавающей точкой (float)
name = "Алексей" # Строка (string)
is_active = True # Логический тип (boolean: либо True, либо False)
```

Функция `type()` помогает узнать, какой тип данных лежит в переменной

```
print(type(age)) # Выведет: <class 'int'>
print(type(name)) # Выведет: <class 'str'>
```

Используйте код с осторожностью.

Базовые типы данных, которые нужно запомнить сразу:

1. **int** (целые числа): 5, -10, 42
2. **float** (дробные числа): 3.14, -0.5
3. **str** (строки): текст внутри одинарных или двойных кавычек
"Привет" или 'Python'
4. **bool** (логический тип): только **True** (Истина) или **False** (Ложь)

Для каких целей применяется каждый тип данных?

- **Целые числа (int)**
 - **Цель:** Счётчики, ID пользователей, возраст, количество товаров на складе.
 - *Пример:* `items_in_cart = 3`
- **Дробные числа (float)**
 - **Цель:** Точные измерения, финансовые расчеты (цены), координаты на карте, проценты.
 - *Пример:* `rating = 4.8`
- **Строки (str)**
 - **Цель:** Любой текст — имена пользователей, email-адреса, сообщения в чате, пароли, пути к файлам.
 - *Пример:* `user_email = "alex@mail.com"`
- **Логический тип (bool)**
 - **Цель:** Флаги-переключатели статуса («Да/Нет»). Нужны для проверки условий (включен ли свет, оплачена ли подписка, залогинен ли пользователь).
 - *Пример:* `is_premium_user = True`

Метод — это встроенное действие, которое можно совершить с конкретным типом данных.

1. **Функция `print()`**

- **Зачем:** Чтобы вывести информацию на экран для программиста или пользователя. Без неё программа отработает в памяти молча.

2. **Функция `type()`**

- **Зачем:** Чтобы провести «диагностику». Когда вы получаете данные из интернета или от пользователя, вы не всегда уверены, что это — текст или число. `type()` помогает программе точно узнать тип данных и не выдать ошибку.

3. **Функции преобразования типов (`int()`, `str()`, `float()`, `bool()`)**

- **Зачем:** Изменять тип «на лету». Например, пользователь ввел на сайте свой возраст текстом "25". Чтобы прибавить к нему 1 год, код должен превратить этот текст в настоящее число с помощью `int("25")`.

условные операторы (`if`, `elif`, `else`). Они нужны для того, чтобы программа могла принимать решения в зависимости от ситуации (прямо как человек).

В **Python** условия строятся на основе логики: «**ЕСЛИ выполнено условие, то делаем ТАК, ИНАЧЕ делаем ПО-ДРУГОМУ**».

- **`if`** (Если) — первая и главная проверка.
- **`elif`** (Сокращение от **`else if`** — А если) — дополнительная проверка, если предыдущее условие не подошло. Их может быть сколько угодно.
- **`else`** (Иначе) — финальный блок, который срабатывает, если **ни одно** из условий выше не выполнилось.

(Проверка баланса на подписку)

```
balance = 400
subscription_price = 499
```

```
if balance >= subscription_price:
```

```
    # Этот код выполнится, если денег хватает
```

```
    print("Подписка успешно продлена!")
```

```
    balance = balance - subscription_price
```

```
elif balance >= 100:
```

```
    # Срабатывает, если денег меньше 499, но больше или
    равно 100
```

```
print("Денег на месяц не хватает. Хотите купить
недельную версию за 99 руб?")
else:
    # Сработает в любом другом случае (если баланс
    меньше 100)
    print("Недостаточно средств. Пополните баланс.")
```

Важные правила синтаксиса Python:

1. **Двоеточие (:)** — обязательно ставится в конце каждой строки с **if**, **elif** и **else**.
2. **Отступы (4 пробела или Tab)** — всё, что находится «внутри» условия, обязательно пишется с отступом. Python по этим отступам понимает, какой кусок кода относится к условию, а какой идет дальше.

Операторы сравнения, которые используются в условиях:

- **>** и **<** (Больше / Меньше)
- **>=** и **<=** (Больше или равно / Меньше или равно)
- **==** (Равно) — **Внимание!** Один знак **=** — это присваивание значения в переменную, а два знака **==** — это сравнение.
- **!=** (Не равно)

Цикл **while** (Пока)

Этот цикл работает до тех пор, пока выполняется определенное условие. Как только условие становится ложным (**False**), цикл останавливается.

Пример из жизни: Телефон заряжается, *пока* батарея не заполнится до 100%.

```
battery = 95
```

```
while battery < 100:
```

```
    battery = battery + 1
```

```
    print(f"Зарядка: {battery}%")
```

```
print("Телефон полностью заряжен!")
```

Что здесь происходит: Код внутри цикла прибавляет по 1 к переменной **battery** и выводит текст. Цикл прокрутится 5 раз (96, 97, 98, 99, 100), после чего условие **100 < 100** станет неверным, и программа пойдет дальше.

Если условие всегда остается верным (например, забыть прибавлять процент зарядки), цикл станет **бесконечным**, и программа "зависнет".

Цикл **for** (Для каждого)

Этот цикл идеален, когда вы заранее знаете, сколько раз нужно повторить действие, или когда нужно перебрать элементы в группе (например, список имен, чисел или букв в строке).

Пример 1: Повторить действие ровно 5 раз с помощью функции `range()`

```
for i in range(5):  
    print("Привет!") # Напечатает "Привет!" ровно 5 раз
```

Перебор элементов (коллекции):

```
for letter in "Python":  
    print(letter)
```

Подсчет суммы чисел (Классическая задача)

С помощью **for** очень удобно что-то считать. Например, найдем сумму всех чисел от 1 до 10.

```
total_sum = 0 # Сюда мы будем складывать результат
```

```
for number in range(1, 11): # Перебираем числа от 1 до 10  
    total_sum = total_sum + number
```

```
print(f"Сумма чисел от 1 до 10 равна: {total_sum}")  
# Выведет: Сумма чисел от 1 до 10 равна: 55
```

Шаг в цикле (Третий аргумент **range**)

Если в **range()** передать три числа, то третье станет **шагом** (через сколько перепрыгивать).

```
# Выведем только четные числа от 2 до 10  
for count in range(2, 11, 2):  
    print(f"Четное число: {count}")
```

В консоли будет: 2, 4, 6, 8, 10.

Что такое список (**list**)?

Представьте себе обычный органайзер для таблеток или поезд с вагонами. Каждый вагон — это ячейка, в которую можно положить данные (числа, строки или даже другие списки).

Создание списка и индексы

Список создается в квадратных скобках **[]**. У каждого элемента есть свой **индекс** (порядковый номер).

Самое главное правило: в программировании отсчет всегда начинается с **0**, а не с **1**.

Создаем список топ-игр или фильмов

```
games = ["Minecraft", "Witcher", "Cyberpunk", "GTA"]
```

Как достать элемент из списка?

```
print(games[0]) # Выведет: Minecraft (первый элемент)
```

```
print(games[2]) # Выведет: Cyberpunk (третий элемент)
```

Если считать с конца, можно использовать минус:

```
print(games[-1]) # Выведет: GTA (самый последний элемент)
```

Изменение и добавление элементов

Списки можно легко менять прямо в процессе работы программы.

```
shopping_list = ["Молоко", "Хлеб"]
```

1. Добавить элемент в КОНЕЦ списка (.append)

```
shopping_list.append("Сыр")
```

Теперь список: ["Молоко", "Хлеб", "Сыр"]

2. Заменить элемент по его индексу

```
shopping_list[1] = "Багет"
```

Заменяли "Хлеб" (индекс 1) на "Багет".

Теперь список: ["Молоко", "Багет", "Сыр"]

Объединяем **for** и **list** (Связка двух тем)

Цикл **for** идеально создан для того, чтобы «пробежаться» по спискам и обрабатывать каждый элемент по очереди.

```
students = ["Аня", "Игорь", "Олег", "Маша"]
```

```
# Цикл по очереди берет каждое имя из списка и кладет в переменную имя (name)
```

```
for name in students:
```

```
    print(f"Привет, {name}! Добро пожаловать на урок Python.")
```

Узнать длину списка — функция `len()`

Показывает, сколько всего элементов находится внутри.

```
items = ["телефон", "ключи", "кошелек"]  
print(len(items)) # Выведет: 3
```

Удалить элемент — метод `.remove()` или `del`

```
items.remove("ключи") # Удалит элемент по его названию  
# Или по индексу: del items[0]
```

Проверить, есть ли элемент в списке — оператор `in`

Идеально подходит для условий `if`

```
cart = ["молоко", "хлеб", "яблоки"]
```

```
if "хлеб" in cart:  
    print("Хлеб уже куплен!")
```

Цикл в цикле (Вложенные циклы)

Это ситуация, когда внутри одного цикла запускается другой. Внешний цикл ждет, пока внутренний не выполнится полностью, и только потом переходит на следующий круг.

- Самый наглядный пример — **вывод сетки координат или таблицы умножения:**

```
# Внешний цикл для строк  
for x in range(1, 4):  
    # Внутренний цикл для столбцов  
    for y in range(1, 4):
```

```
print(f"[{x}:{y}]", end=" ") # end=" " чтобы печатать в одну строку
print() # Перенос строки после завершения внутреннего цикла
```

```
[1:1] [1:2] [1:3]
[2:1] [2:2] [2:3]
[3:1] [3:2] [3:3]
```

Продвинутые методы списков (**list**)

Методы — это встроенные в Python функции, которые умеют делать со списками сложные вещи в одну строчку кода.

```
numbers = [5, 2, 9, 1, 5]
```

1. Сортировка по возрастанию (.sort)

```
numbers.sort() # numbers станет: [1, 2, 5, 5, 9]
```

2. Перевернуть список задом наперед (.reverse)

```
numbers.reverse() # numbers станет: [9, 5, 5, 2, 1]
```

3. Посчитать, сколько раз элемент встречается в списке (.count)

```
print(numbers.count(5)) # Выведет: 2 (число 5 встречается дважды)
```

4. Узнать индекс (позицию) первого совпадения (.index)

```
print(numbers.index(9)) # Выведет: 0 (девятка теперь на первом месте)
```

5. Вставить элемент в конкретное место (.insert)

(на место с индексом 1 вставляем число 99)

```
numbers.insert(1, 99) # Список станет: [9, 99, 5, 5, 2, 1]
```

Кортежи: тип данных **tuple**

Кортеж (tuple) — это родной брат списка. Он выглядит почти так же, но создается в **круглых скобках ()**.

Главное отличие: **кортеж НЕЛЬЗЯ ИЗМЕНИТЬ**. Нельзя добавить элементы, удалить или поменять их местами. Его называют "read-only список".

Создаем кортеж (например, неизменные координаты города)

```
coordinates = (55.7558, 37.6173)
```

`print(coordinates[0])` # Выведет: 55.7558

Попытка изменить вызовет ошибку (TypeError):
`coordinates[0] = 60.0` # ТАК ДЕЛАТЬ НЕЛЬЗЯ, программа сломается!

Зачем он нужен? Он защищает данные от случайных изменений и работает в памяти компьютера чуть быстрее, чем обычный список.

модуль `os` и функция `os.walk()`

Это мощный инструмент для работы с вашей операционной системой (Windows/Mac/Linux). Модуль `os` позволяет Python «смотреть» на папки и файлы на вашем компьютере. Функция `os.walk()` — это «прогулка» по папкам. Она заходит в указанную папку, смотрит, что в ней лежит, заходит во все подпапки и выдает нам их содержимое.

Пример: Поиск всех файлов в папке

`import os` # Обязательно импортируем модуль перед использованием

Укажем путь к какой-нибудь папке (например, текущая папка проекта '.')
`root_path = '.'`

`os.walk` возвращает три вещи на каждом шагу:
имя папки, список подпапок внутри неё и список файлов
`for current_dir, subdirs, files in os.walk(root_path):`
 `print(f"Сейчас я нахожусь в папке: {current_dir}")`

Переберем файлы в этой папке
`for file in files:`
 `print(f" -> Нашел файл: {file}")`

Этот скрипт часто используют, чтобы написать своего робота, который автоматически сортирует фотографии на компьютере, чистит корзину или ищет затерявшиеся документы.

Где и зачем используются Кортежи (`tuple`)?

Основная фишка кортежа — **неизменяемость** и **безопасность**. Если вы записали туда данные, ни один баг в программе не сможет их случайно перезаписать.

В играх и программах это применяется постоянно:

- **Координаты и сетка (Зафиксировать положение):**

В 2D и 3D играх координаты игрока или объектов часто хранятся в кортежах.

Позиция игрока на карте: (X, Y)

player_position = (10, 25)

Цвета в формате RGB (Красный, Зеленый, Синий)

ENEMY_COLOR = (255, 0, 0) # Чистый красный, он не должен измениться в ходе игры

Пример: Вы делаете шахматы. Доска — это фиксированные клетки от (0, 0) до (7, 7). Кортежи идеально подходят для обозначения этих клеток.

Конфигурация и настройки (Что-то запечатлеть):

Разрешение экрана вашего монитора. Оно задается один раз при запуске игры.

SCREEN_RESOLUTION = (1920, 1080)

Возврат нескольких значений из функции:

Если вашей функции нужно выдать сразу несколько ответов (например, имя игрока и его здоровье), Python объединяет их в кортеж.

def get_player_stats():

name = "Warrior"

hp = 100

return name, hp # Python автоматически вернет это как кортеж ("Warrior", 100)

Как `os.walk` применяется в реальной разработке?

Представьте, что вы написали крутую игру или плеер. Вы же не будете вручную прописывать в коде пути к каждой картинке, звуку или модулю? Игра должна сама «посмотреть» в свою папку и всё загрузить. Вот тут и нужна «прогулка» **`os.walk`**.

- **Автоматическая загрузка ресурсов (моддинг и текстуры):**

Вы делаете игру (например, аналог Minecraft или RPG), которая поддерживает моды или кастомные скины. Игроки закидывают свои файлы в папку **`mods/`**.

С помощью **`os.walk`** ваша игра при запуске сканирует эту папку,

находит все файлы с расширением `.png` (картинки) или `.json` (настройки мода) и автоматически добавляет их в игру.

- **Система сохранений (Save/Load):**

Игрок нажимает кнопку «Загрузить игру». Программа с помощью `os.walk` заходит в папку `saves/`, считывает названия всех файлов сохранений и красиво выводит их списком в меню игры:

1. `Save_Auto_1.dat`
2. `Save_Before_Boss.dat`

- **Создание плеера (Сбор файлов):**

Вы пишете музыкальный плеер или галерею. Пользователь указывает папку со своей музыкой. `os.walk` пробегается по всем подпапкам, находит файлы `.mp3` и составляет из них плейлист для вашей программы.

Подведем итог на простом примере игры:

1. Вы используете `os.walk`, чтобы игра «осмотрелась» в папке `assets/sounds/`, нашла все звуковые файлы и загрузила их в память.
2. Когда монстр бьет игрока, игра проверяет его фиксированный тип урона из кортежа `MONSTER_DAMAGE = (15, "fire")`, который защищен от случайного изменения (чтобы огонь вдруг не превратился в лед из-за бага).

Функции (`def`)

Это не просто «еще один элемент». Это инструмент, который создан специально для того, чтобы красиво упаковывать и объединять всё, что вы выучили до этого (переменные, условия, циклы и списки) в удобные, аккуратные блоки

Функции — это способ дать ленивому компьютеру четкую инструкцию один раз, назвать её понятным словом, а потом вызывать её одной строчкой кода в любом месте программы.

Представьте, что функция — это кулинарный рецепт. Слово `def` (от английского **define** — определить) означает, что мы этот рецепт записываем.

Определение и вызов функций:

Самая простая функция не требует никаких данных на входе, она просто выполняет заготовленный кусок кода.

1. ОПРЕДЕЛЯЕМ функцию (создаем рецепт)

```
def greet_user():  
    print("Привет, игрок!")  
    print("Загрузка вашего сохранения...")
```

---- Код ниже пока ничего не выведет, функция просто лежит в памяти ----

2. ВЫЗЫВАЕМ функцию (начинаем готовить по рецепту)

greet_user() # Выведет два текста на экран

greet_user() # Можно вызвать еще раз, код повторится!

Параметры и аргументы:

Функции становятся по-настоящему мощными, когда мы можем передавать им разные данные для обработки.

- **Параметры** — это переменные-заглушки, которые мы пишем в скобках при создании функции.
- **Аргументы** — это реальные значения, которые мы подставляем туда при вызове.

Нам нужен калькулятор урона для игры. В скобках пишем параметры:

def calculate_damage(base_attack, weapon_bonus):

total = base_attack + weapon_bonus

print(f"Вы нанесли {total} единиц урона!")

Передаем реальные аргументы для Мечника:

calculate_damage(50, 15) # Выведет: Вы нанесли 65 единиц урона!

Передаем реальные аргументы для Лучника:

calculate_damage(30, 40) # Выведет: Вы нанесли 70 единиц урона!

Возврат значения: оператор **return**

Функция может не просто выводить текст на экран (**print**), а **возвращать** результат своей работы обратно в программу, чтобы мы могли сохранить его в переменную.

def double_number(x):

return x * 2 # Функция отдает результат
обратно

```
# Сохраняем результат работы функции в переменную
```

```
my_coins = double_number(50)
```

```
print(my_coins)    # Выведет: 100
```

Переменное количество аргументов: параметр ***args**

Что делать, если мы не знаем заранее, сколько данных пользователь передаст в функцию? Например, игрок может купить в магазине 2 предмета, а может 5, и нам нужно посчитать общую стоимость. Звездочка ***** перед именем параметра превращает все переданные аргументы в **кортеж**

```
# Считаем сумму всех переданных чисел (сколько бы их ни было)
```

```
def sum_all_prices(*args):
```

```
    # args внутри функции — это кортеж,
    # например: (100, 250, 50)
```

```
    total = 0
```

```
    for price in args:    # Комбинируем функцию с
                           # циклом for!
```

```
        total += price
```

```
    return total
```

```
# Вызываем с разным количеством товаров:
```

```
print(sum_all_prices(100, 50))    # Выведет:
150 (2 товара)
```

```
print(sum_all_prices(10, 20, 30, 40)) # Выведет:
100 (4 товара)
```

Звездочка ***** перед **args** сообщает Python: «Собери все аргументы, которые введет пользователь (сколько бы их ни было — 2, 5 или 100), и упакуй их в один неизменяемый мешок (кортеж)»

```
def buy_items(*args):
```

```
    # В этот момент Python уже зафиксировал всё, что выбрал игрок.
```

```
    # Если игрок купил Меч, Щит и Зелье, то args превращается в кортеж:
```

```
    # args = ("Меч", "Щит", "Зелье")
```

```
    print(f"Игрок положил в корзину товаров: {len(args)}") #  
    # Посчитает количество
```

```
    for item in args:
```

```
        print(f"Робот на кассе сканирует: {item}")
```

```
# Игрок 1 берет 2 предмета:
```

```
buy_items("Лук", "Стрелы")
```

```
# Игрок 2 берет 4 предмета:
```

```
buy_items("Шлем", "Броня", "Сапоги", "Перчатки")
```

То есть ***args** — это универсальный «приемник» для любого количества вещей. Он фиксирует их внутри функции в виде кортежа, и мы можем делать с ними что угодно: считать количество через **len()**, складывать цены или перебирать списком.

1. Кортежи (**tuple**) — это «Законы физики» игры

Они отвечают за константы и жесткие правила, которые никто не может изменить в процессе игры (чтобы не сломать баланс).

- В коде игры: Цены на товары в магазине и их вес. Если Меч стоит 100 золотых, код не должен случайно переписать эту цену во время боя.

2. Списки (**list**) — это «Инвентарь и изменяемый мир»

Они отвечают за динамику. Это вещи, которые постоянно меняются: количество предметов в рюкзаке, список врагов на карте, список активных квестов.

- В коде игры: Рюкзак твоего персонажа. Ты поднял предмет — список увеличился. Выбросил — уменьшился.

3. Модуль **os** и функция **os.walk** — это «Загрузчик и память»

Они отвечают за **связь игры с реальным компьютером**. Игра не знает, какие сохранения или моды лежат на жестком диске, пока не «осмотрится» с помощью этого инструмента.

- *В коде игры:* При запуске сканирует папку **saves/**, находит файл **save_level_5.json** и загружает твоего персонажа.

4. Параметр *args в Функциях — это «Универсальный приемщик» на кассе

Он отвечает за **гибкость при передаче данных**. Когда игрок совершает действие с неизвестным количеством предметов (выбрасывает 3 зелья или покупает 5 свитков), ***args** ловит их все за один раз.

1. КОРТЕЖИ: Фиксированные цены товаров (Предмет, Цена).
Изменить нельзя!

```
PRICE_SWORD = ("Меч", 150)
PRICE_SHIELD = ("Щит", 100)
PRICE_POTION = ("Зелье", 50)
```

2. СПИСКИ: Рюкзак игрока. Сначала он пустой, но будет меняться.
player_inventory = ["Старая куртка"]
player_gold = 300

4. ФУНКЦИЯ И *ARGS: Универсальная касса.

Звездочка поймает любое количество купленных товаров в один мешок.

```
def buy_items_manager(*items_to_buy):
    global player_gold # Позволяет функции менять золото снаружи
```

```
    print(f"--- Оформление покупки ---")
```

```
    # Цикл for перебирает мешок *args
```

```
    for item in items_to_buy:
```

```
        name = item[0] # Берем имя из кортежа
```

```
        price = item[1] # Берем цену из кортежа
```

```
        if player_gold >= price:
```

```
            player_gold -= price
```

```
            player_inventory.append(name) # Добавляем в СПИСОК
```

```
            print(f"Куплено: {name} за {price} золотых.")
```

```
        else:
```

```
            print(f"Не хватает золота на {name}!")
```

```
# 3. МОДУЛЬ OS (Имитация): Игра проверяет, есть ли папка для
сохранений
if not os.path.exists("game_saves"):
    os.makedirs("game_saves") # Создаем папку на компьютере, если
её нет
```

```
# --- ИГРОВОЙ ПРОЦЕСС (Вызов кода) ---
```

```
# Игрок пришел к торговцу и покупает СРАЗУ 3 предмета. *args их
зафиксирует!
buy_items_manager(PRICE_SWORD, PRICE_POTION,
PRICE_SHIELD)
```

```
# Проверяем, что изменилось в изменяемом СПИСКЕ инвентаря
print(f"\nОстаток золота: {player_gold}")
print(f"Ваш рюкзак сейчас: {player_inventory}")
```

Область видимости переменных (Scope)

область видимости — это зона в коде, внутри которой переменная «жива», и программа её видит. В Python есть два главных лагеря: **глобальные** и **локальные** переменные.

Представьте многоквартирный дом. Глобальная переменная — это то, что висит на улице (номер дома). Локальная переменная — это вещи внутри конкретной квартиры (функции). Соседи с улицы не знают, что лежит у вас в тумбочке, пока вы сами это не вынесете.

1. Локальная область видимости (Local Scope)

Переменная, созданная **внутри** функции, является локальной. Она рождается в момент вызова функции и полностью «умирает», когда функция заканчивает работу. Снаружи кода к ней обратиться **нельзя**.

```
def spawn_enemy():
    enemy_health = 100 # ЛОКАЛЬНАЯ переменная
    print(f"Здоровье врага внутри функции: {enemy_health}")
```

```
spawn_enemy() # Вызов работает отлично
```

```
# А теперь пытаемся узнать здоровье врага вне функции:
print(enemy_health) # ОШИБКА! NameError: name 'enemy_health' is not
defined
```

Почему это круто: Разные функции могут использовать одинаковые имена переменных внутри себя, и они никогда не перепутаются и не испортят код друг друга.

Глобальная область видимости (**Global Scope**)

Переменная, созданная в основном тексте программы (вне любых функций), называется глобальной. Её видят **все** функции в этом файле.

```
player_name = "Arthas" # ГЛОБАЛЬНАЯ переменная
```

```
def show_profile():
```

```
    # Функция спокойно "видит" имя игрока снаружи
```

```
    print(f"Игрок: {player_name}")
```

```
show_profile() # Выведет: Игрок: Arthas
```

Главная ловушка: Попытка изменить глобальное внутри локального

Смотреть на глобальную переменную функция может, а вот **изменять её напрямую — нет**. Если вы попытаетесь перезаписать её внутри функции, Python решит, что вы просто создаете новую локальную переменную с таким же именем.

```
gold = 500 # Глобальное золото
```

```
def подбор_монеты():
```

```
    gold = gold + 100 # ОШИБКА! Python запутается и выдаст  
UnboundLocalError
```

Как это решить правильно? (Два пути)

- **Путь А: Через `return` (Самый правильный и профессиональный)**

Функция принимает значение, считает результат и отдает его обратно в основной код.

```
gold = 500
```

```
def add_gold(current_gold):
```

```
    return current_gold + 100 # Просто возвращаем результат вычислений
```

```
gold = add_gold(gold) # Перезаписываем глобальную переменную  
снаружи  
print(gold) # Выведет: 600
```

Путь Б: Ключевое слово **global**

Вы жестко приказываете функции: «Работай напрямую с уличной (глобальной) переменной!».

```
gold = 500
```

```
def cheat_code():
```

```
    global gold # Говорим, что будем менять именно внешнюю  
    переменную gold
```

```
    gold = 99999
```

```
cheat_code()
```

```
print(gold) # Выведет: 99999
```

- **Совет разработчика:** Инструментом **global** лучше не злоупотреблять. Если в большой программе сотни функций будут одновременно менять одну и ту же переменную, начнется хаос и полезут труднодобываемые баги.

В обычных программах (не играх) эти темы применяются для управления данными пользователей, работы с серверами, настройками и интерфейсом.

- **Локальные переменные (Local Scope):** Используются внутри функции, которая, например, проверяет правильность введенного пароля или вычисляет сумму скидки в корзине интернет-магазина. Как только расчет окончен, эти временные данные стираются из памяти.
- **Глобальные переменные (Global Scope):** Хранят информацию о текущем авторизованном пользователе (например, **CURRENT_USER_ID = 452**), токен доступа к серверу или язык интерфейса (русский/английский), который должен быть доступен во всех окнах программы.

- **Структура кода:** Позволяет разбивать большие утилиты на понятные блоки — функция отправки Email, функция выгрузки отчета в Excel, функция авторизации.

Словари: тип данных **dict** и их методы

Словарь (dict) — это невероятно мощный тип данных. Если в списке элементы хранятся под номерами (0, 1, 2...), то в словаре данные хранятся в формате **Ключ : Значение**.

Представьте настоящую анкету, телефонную книгу или карточку товара в интернет-магазине. Вместо индексов мы используем понятные слова-метки. Словари создаются в фигурных скобках **{ }**.

Создание словаря и доступ к данным:

Создаем карточку пользователя (анкету)

```
user_profile = {  
    "name": "Иван Иванович",  
    "age": 28,  
    "email": "ivan@example.com",  
    "is_premium": True  
}
```

Как достать значение? Пишем имя ключа в квадратных скобках:

```
print(user_profile["name"]) # Выведет: Иван Иванович  
print(user_profile["email"]) # Выведет: ivan@example.com
```

Изменение и добавление данных

Словари, как и списки, можно легко менять на лету. Если ключа нет — он создастся, если есть — перезапишется.

Изменяем существующее значение (День рождения прошел)

```
user_profile["age"] = 29
```

Добавляем абсолютно новый ключ с данными

```
user_profile["city"] = "Москва"
```

```
print(user_profile)
```

Внутри будет: {'name': 'Иван...', 'age': 29, 'email': '...', 'is_premium': True, 'city': 'Москва'}

Важнейшие методы словарей (**dict**)

Чтобы эффективно управлять такими анкетами, в Python есть встроенные методы:

- **Безопасное чтение — `.get()`**

Если попытаться вызвать ключ, которого нет (например, `user_profile["phone"]`), программа вылетит с ошибкой. Метод `.get()` вернет **None** или ваш стандартный ответ, если ключа не существует.

Ключа "phone" нет, поэтому программа не сломается, а выведет: "Номер не указан"

```
user_phone = user_profile.get("phone", "Номер не указан")
```

Получить только ключи — **`.keys()`**

```
print(user_profile.keys()) # Выведет список всех "заголовков": ['name', 'age', 'email'...]
```

Получить только значения — **`.values()`**

```
print(user_profile.values()) # Выведет только чистые данные: ['Иван Иванович', 29...]
```

Получить пары ключ-значение — **`.items()`**

Этот метод идеален для комбинации с циклом **for**, когда нужно красиво напечатать всю анкету на экране.

```
for key, value in user_profile.items():
```

```
    print(f"Поле {key}: {value}")
```

Словари — это фундамент для работы с базами данных и веб-сайтами.

словари и их методы — это фундамент для работы с любыми структурированными данными, от профилей пользователей до ответов от серверов.

Вот конкретные примеры, какую выгоду и цели преследуют программисты, используя методы словарей в реальных программах и веб-сайтах:

1. Метод `.get()` — Спасение от вылетов программы (Выгода: Стабильность)

Представьте, что вы делаете форму регистрации или личный кабинет. Пользователь заполнил профиль, но **не указал номер телефона** (это поле было необязательным).

Если ваша программа попытается напрямую вызвать `user["phone"]`, она мгновенно **вылетит с ошибкой (Crash)** у тысяч пользователей.

- **Цель:** Безопасно проверить данные.
- **Применение в коде:**

```
# Вместо падения программы мы даем ей "план Б"
```

```
phone = user_profile.get("phone", "Телефон не привязан")
```

```
print(f"Связь с клиентом: {phone}")
```

```
# Если телефона нет, программа не сломается, а просто напишет  
"Телефон не привязан"
```

Метод `.keys()` — Умный поиск и фильтрация (Выгода: Скорость и Автоматизация)

Представьте, что вы делаете **интернет-магазин** одежды. У вас есть товар со словарём характеристик: `{"размер": "XL", "цвет": "красный", "материал": "хлопок"}`.

Пользователь вбивает в поиск слово **"скидка"**. Вам нужно быстро проверить, есть ли вообще такое свойство (ключ) у данного товара, чтобы показать нужный значок.

- **Цель:** Быстро вытащить все заголовки (характеристики) для проверки.
- **Применение в коде:**

```
# Проверяем, настроена ли акция для товара
```

```
if "discount_price" in product.keys():  
    print("Показать ценник со скидкой!")
```

Метод `.values()` — Подсчет аналитики (Выгода: Финансовый учет и Статистика)

Вы пишете CRM-систему для автосервиса или учета финансов. У вас есть словарь, где ключ — это имя мастера, а значение — сколько денег он заработал за день: `{"Олег": 5000, "Антон": 7500, "Мария": 9000}`.

Вам, как владельцу бизнеса, в конце дня не важны имена (ключи), вам нужна **общая сумма** выручки, чтобы посчитать чистую прибыль.

- **Цель:** Игнорировать текст и собрать только цифры для математических операций.
- **Применение в коде:**

```
salaries = {"Олег": 5000, "Антон": 7500, "Мария": 9000}
```

```
# Метод .values() отдает чистые числа, и мы сразу передаем их в
функцию суммы sum()
total_payout = sum(salaries.values())
print(f"Всего выплачено сотрудникам: {total_payout} руб.") # Выведет:
21500
```

Метод `.items()` — Красивый вывод и генерация документов (Выгода: Удобство для людей)

Компьютер обожает фигурные скобки и кавычки словарей, но обычному человеку (вашему клиенту) такую «кашу» показывать нельзя. Метод `.items()` берет каждую пару «Ключ: Значение» и аккуратно разделяет их, чтобы вы могли построить красивый интерфейс.

- **Цель:** Превратить сырые данные из базы в читаемый текст на экране, в чеке или в PDF-отчете.
- **Применение в коде:**

```
order = {"Товар": "Ноутбук", "Цена": 75000, "Доставка": "Курьер"}
```

```
print("--- Ваш чек ---")
# Метод .items() раскладывает пару на две понятные переменные
for характеристика, значение in order.items():
    print(f"{характеристика}: {значение}")
```

```
--- Ваш чек ---
Товар: Ноутбук
Цена: 75000
Доставка: Курьер
```

Словари позволяют хранить данные в идеальном порядке (как папки в архиве), а методы `.get()`, `.keys()`, `.values()` и `.items()` — это ваши «инструменты-манипуляторы», с помощью которых вы заставляете эти данные приносить коммерческую или техническую пользу вашей программе.

Чтение и запись файлов

До этого момента все наши данные (переменные, списки, словари) жили только в оперативной памяти компьютера. Стоило закрыть программу — и они исчезали. Файлы позволяют сохранять информацию навсегда на жесткий диск: записывать отчеты, сохранять прогресс или читать настройки.

В Python для этого используется встроенная функция `open()`.

Как устроен базовый процесс

Чтобы поработать с файлом, программа выполняет три шага:

Открыть ➔ Сделать дело ➔ Обязательно Закрыть (иначе файл заблокируется в системе).

При открытии нужно указать **режим работы**:

- `'w'` (*write*) — запись. **Внимание:** полностью стирает старое содержимое файла и пишет заново!
- `'a'` (*append*) — добавление. Дописывает новые данные в конец файла, сохраняя старые.
- `'r'` (*read*) — чтение файла.

1. Открываем файл text.txt в режиме записи ('w')

```
file = open("text.txt", "w", encoding="utf-8") # encoding нужен, чтобы русский язык отображался корректно
```

2. Пишем текст

```
file.write("Привет! Это первая строка.\n") # \n - это перенос на новую строку
```

3. Закрываем файл! (Очень важно)

```
file.close()
```

Менеджер контекста `with ... as` (Идеальный подход)

Конструкция `with` автоматически закрывает файл сама, даже если внутри программы произойдет ошибка или вылетит баг. Это защищает жесткий диск от утечек памяти.

```
# Открываем файл в режиме добавления ('a')
with open("logs.txt", "a", encoding="utf-8") as file:
    file.write("Пользователь зашел в систему\n")
    file.write("Нажата кнопка 'Скачать отчет'\n")
# Как только блок кода закончился (сместился отступ) — Python SAM
закрыл файл!
```

Чтение данных из файла:

Представьте, что программа при старте должна прочитать список заблокированных пользователей из текстового файла.

```
with open("logs.txt", "r", encoding="utf-8") as file:
    # Метод .read() считывает ВСЬ файл целиком в одну строку
    content = file.read()
    print(content)
```

Или построчное чтение (если файл огромный и нужно сэкономить память):

```
with open("logs.txt", "r", encoding="utf-8") as file:
    # Цикл for умеет читать файл построчно
    for line in file:
        print(f"Запись из логов: {line.strip()}") # .strip() убирает лишние
пустые отступы
```

Где это применяется в программах (Выгода и Цели)

- **Логирование (Logs):** Любая серьезная программа (хоть Telegram, хоть банковский софт) тайно записывает в файл `logs.txt` всё, что она делает каждую секунду. Если у пользователя что-то сломается, программист попросит этот файл и по записям поймет, в чем была ошибка.
- **Импорт/Экспорт данных:** Выгрузка базы данных клиентов в формат `.csv` или `.txt` для бухгалтерии.
- **Сохранение конфигураций:** Файлы настроек вроде `settings.txt`, где программа хранит выбранную тему оформления (темная/светлая) или путь, куда скачивать файлы.

Множества: тип данных `set`

Множество (set) — это коллекция, у которой есть два уникальных свойства:

1. В ней **все элементы абсолютно уникальны** (дубликаты автоматически удаляются).

2. В ней **нет порядка (индексов)** — элементы перемешаны как ягоды в корзине.

Создается множество тоже в фигурных скобках { }, но, в отличие от словаря, здесь пишутся просто одиночные элементы через запятую, без двоеточий.

1. Главный фокус: Мгновенное удаление дубликатов (Выгода для программ)

Представьте, что вы делаете систему регистрации. Пользователи вводили свои Email, и из-за сбоя некоторые адреса продублировались. Вам нужно получить чистый список уникальных почт клиентов для рассылки.

Сырой список из базы данных с кучей повторений

```
dirty_emails = ["user1@mail.ru", "admin@test.ru", "user1@mail.ru",  
"boss@work.ru", "admin@test.ru"]
```

Превращаем список во множество функцией set() -> магия! Все дубликаты исчезают сами

```
clean_emails = set(dirty_emails)
```

```
print(clean_emails)
```

Выведет только уникальные: {'admin@test.ru', 'user1@mail.ru', 'boss@work.ru'}

2. Математические операции (Цель: Аналитика данных)

Множества умеют мгновенно сравнивать группы данных между собой.

В Python для этого есть специальные знаки:

- **Пересечение (&)** — найти общие элементы.

Пример в программах: Найти пользователей, которые одновременно скачали и мобильное приложение, и зашли через компьютер.

```
pc_users = {"Иван", "Мария", "Олег"}
```

```
mobile_users = {"Антон", "Мария", "Елена"}
```

Кто сидит и там, и там?

```
cross_users = pc_users & mobile_users
```

```
print(cross_users) # Выведет: {'Мария'}
```

Разность (-) — найти уникальные элементы одной группы.

Пример в программах: У вас есть список всех товаров магазина и список проданных товаров. Вычитаем одно из другого — получаем то, что осталось на складе.

```
all_goods = {"хлеб", "молоко", "сыр", "яблоки"}
```

```
sold_goods = {"хлеб", "сыр"}
```

```
in_stock = all_goods - sold_goods  
print(in_stock) # Выведет: {'молоко', 'яблоки'}
```

Строки и их методы (**str**)

Python умеет не просто хранить строки, а мощно ими манипулировать: чистить от мусора, менять регистр и красиво форматировать.

Экранированные символы

Иногда внутри текста нужно использовать символы, которые Python считает системными (например, кавычки или перенос строки). Для этого используется косая черта `\` (обратный слэш) — она «экранирует» следующий за ней символ.

- `\n` — перенос текста на новую строку.
- `\t` — табуляция (красная строка / большой отступ).
- `\'` или `\"` — позволяют вставить кавычку внутрь строки, чтобы Python не подумал, что текст закончился.

Пример использования разных символов

```
text = "Привет!\n\tЭто строка с \"отступом\" и переносом."  
print(text)
```

Привет!

Это строка с "отступом" и переносом.

Топовые методы строк (**str**)

Методы строк позволяют мгновенно обрабатывать текст. Представьте, что пользователь вводит свой Email при авторизации: один написал капсом, другой случайно поставил пробелы по бокам. Методы решают эту проблему:

- `.strip()` — Удаление пробелов по краям (Выгода: Защита от скрытых ошибок).

Часто пользователи случайно нажимают пробел в конце ввода. Для компьютера "ЛОГИН" и "ЛОГИН " — это разные строки. `.strip()` стирает невидимые пробелы.

- `.lower()` и `.upper()` — Приведение к регистру.

Делает все буквы маленькими или большими. Полезно, чтобы проверять промокоды без учета регистра (чтобы скидка50 и СКИДКА50 сработали одинаково).

- **`.replace()` — Замена одного текста на другой.**

```
phone = "+7 (999) 111-22-33"
```

```
# Очищаем номер от скобок и тире для отправки на сервер
```

```
clean_phone = phone.replace(" ", "").replace("(", "").replace(")", "").replace("-", "")
```

```
print(clean_phone) # Выведет: +79991112233
```

`.split()` — Разбить строку на список.

Разрезает строку по указанному разделителю (например, по пробелу или запятой).

```
tags = "python,программирование,учеба"
```

```
tags_list = tags.split(",") # Превратится в список: ['python', 'программирование', 'учеба']
```

f-строки и форматирование

f-строки (форматированные строки) — это самый удобный, современный и красивый способ объединять текст и переменные. Вы просто ставите букву **f** перед кавычками, а внутри текста пишете переменные в фигурных скобках **{ }**.

- **Цель:** Конструировать динамические сообщения для пользователей.

```
user = "Алексей"
```

```
bonus_points = 250
```

```
# Старый неудобный способ (через плюс):
```

```
# message = "Привет, " + user + "! У вас " + str(bonus_points) + " бонусов."
```

```
# Новый крутой способ (f-строка):
```

```
message = f"Привет, {user}! У вас {bonus_points} бонусов. Напишите промокод в формате LOWER."
```

```
print(message)
```

Где это применяется в реальных программах?

- **Валидация форм:** Проверка, правильно ли введен Email (перевод в **.lower()** и очистка через **.strip()**).
- **Поисковые фильтры:** Когда вы ищете товар, программа переводит ваш запрос в нижний регистр, чтобы найти совпадения

независимо от того, как вы написали слово — с большой или маленькой буквы.

- **Генерация уведомлений:** Создание персональных писем, чеков или SMS-сообщений для клиентов с помощью **f-строк**.

Обработка исключений (**try, except**)

В программировании ошибки делятся на два типа. Первые — это синтаксические (когда вы опечатались в коде), их Python находит сразу и не запускает программу. Вторые — это **исключения (ошибки выполнения)**. Код написан идеально, но во время работы что-то пошло не так: пропал интернет, пользователь ввел буквы вместо цифр или файл на диске был случайно удален.

Если не обработать такое исключение, программа мгновенно **вылетит с ошибкой (закроется)**. Конструкция **try/except** — это «подушка безопасности», которая ловит ошибку, не давая программе упасть.

Как устроен блок **try ... except**

Вы приказываете Python: «**Попробуй (try)** выполнить этот код. Если внутри него случится авария, не паникуй, а плавно перейди в блок **исключения (except)** и сделай то, что там написано».

Пример: Деление на ноль (классическая ошибка)

```
# Опасный кусок кода, где может возникнуть ошибка
number = int(input("Введите число: "))
result = 100 / number
print(f"Результат: {result}")
```

except ZeroDivisionError:

```
# Этот блок работает ТОЛЬКО если пользователь ввел 0
print("Ошибка: На ноль делить нельзя!")
```

except ValueError:

```
# Этот блок работает, если пользователь ввел буквы вместо цифр
print("Ошибка: Вы ввели не число!")
```

Программа не закрылась аварийно. Она вежливо объяснила пользователю, в чем он не прав, и продолжила работать дальше.

Перехват любых ошибок и блок **else / finally**

Конструкцию можно расширить для максимального контроля:

try:

```
file = open("settings.txt", "r")
content = file.read()
except Exception as error:
    # Перехватит ВООБЩЕ любую ошибку и запишет ее описание в
    # переменную error
    print(f"Что-то пошло не так: {error}")
else:
    # Сработает ТОЛЬКО если в блоке try ВСЁ прошло идеально и
    # ошибок НЕ БЫЛО
    print("Файл успешно прочитан!")
finally:
    # Сработает В ЛЮБОМ СЛУЧАЕ (была ошибка или нет).
    # Идеально подходит, чтобы закрыть файл или отключиться от базы
    # данных.
    print("Блок очистки памяти завершен.")
```

Где и зачем это применяется в программах (Выгода и Цели)

- **Работа с интернетом и API:** Вы пишете программу погоды. Она отправляет запрос на сервер. Если у пользователя отвалился Wi-Fi, сервер не ответит. Без **try/except** приложение закроется. С ним — программа напишет: *"Проверьте подключение к сети"*.
- **Защита ввода данных (Защита от дурака):** Люди постоянно ошибаются при заполнении форм (вводят текст в поле для возраста, ставят лишние знаки в номере телефона). **try/except** перехватывает эти данные и просит ввести их заново.
- **Работа с базами данных и файлами:** Если база данных временно перегружена или файл заблокирован другой программой, ваш софт подождет пару секунд и попробует снова, вместо того чтобы полностью отключиться.

Импорт модулей и таинственная конструкция **if __name__ == '__main__':**

Когда ваша программа разрастается, писать весь код в одном файле становится невозможно — получится огромная «портянка» на тысячи строк, в которой легко запутаться. Проекты делят на отдельные модули (файлы **.py**), а затем импортируют их друг в друга. Конструкция **if __name__ == '__main__':** — это главный диспетчер и «входная дверь» в вашу программу.

Как работает импорт модулей

Представьте, что вы написали крутые функции для работы с игроком в файле `player.py`, а основную логику игры хотите вести в файле `main.py`.

- **Файл `player.py` (наш модуль):**

```
def heal(hp):  
    return hp + 20
```

Файл `main.py` (основная программа):

Мы можем подключить файл `player.py` с помощью команды `import`:

```
import player # Подключаем наш модуль
```

```
current_hp = 50
```

Вызываем функцию через имя модуля:

```
current_hp = player.heal(current_hp)  
print(f"Здоровье игрока: {current_hp}") # Выведет: 70
```

Зачем нужна конструкция `if __name__ == '__main__':`?

Когда Python импортирует модуль (например, делает `import player`), он **автоматически выполняет весь код внутри этого модуля сверху вниз**.

И вот тут возникает проблема. Представьте, что внутри файла `player.py` вы ради теста написали вызовы функций или проверку кода. Если просто импортировать этот файл в `main.py`, все ваши тесты мгновенно запустятся и начнут спамить в консоль основного проекта!

Чтобы этого избежать, используют встроенную переменную `__name__` (имя).

- Если файл запускается **напрямую пользователем** (как главный), Python присваивает переменной `__name__` значение `"__main__"`.
- Если файл **импортируется** другим скриптом, переменная `__name__` будет равна названию самого файла (например, `"player"`).

Пример защиты кода:

Запишем файл `player.py` правильно:

```
def heal(hp):  
    return hp + 20
```

Этот блок выполнится ТОЛЬКО если мы запустим сам файл player.py напрямую.

Если мы импортируем этот файл в main.py, этот код будет проигнорирован!

```
if __name__ == '__main__':  
    print("--- ТЕСТИРОВАНИЕ МОДУЛЯ PLAYER ---")  
    test_hp = 10  
    print(f"Тест лечения: {heal(test_hp)}") # Должно быть 30
```

Где и зачем это применяется в реальной разработке (Выгода и Цели)

1. **Точка входа (Main Entry Point):** В профессиональных программах и веб-сервисах конструкция `if __name__ == '__main__':` указывает компьютеру: «Вот отсюда нужно начинать запуск всего приложения!». Обычно туда помещают запуск главного окна, подключение к базе данных или старт игрового цикла.
2. **Тестирование:** Разработчики пишут тесты для своих функций прямо внизу файла под этой конструкцией. Это позволяет проверить работу отдельного модуля на месте, не ломая основное приложение.
3. **Порядок и Архитектура:** Код разделен на чистые библиотеки. Вы точно знаете, что импортируемые модули не начнут совершать самостоятельных действий без вашей команды.

ООП (Объектно-Ориентированное Программирование)

В базе мы использовали словари, чтобы описать объект (например, `player = {"name": "Воин", "hp": 100}`). Но словарь не умеет совершать действия.

ООП позволяет создать **Класс (class)** — это чертеж или шаблон, по которому создаются объекты. У них есть свойства (данные) и методы (функции внутри класса).

• Пример: Создаем персонажей для игры

```
class Hero:
```

`__init__` - это конструктор. Он вызывается при создании каждого нового героя

```
    def __init__(self, name, hp, damage):  
        self.name = name      # Свойство: имя  
        self.hp = hp          # Свойство: здоровье  
        self.damage = damage  # Свойство: урон
```

Метод класса: действие, которое герой умеет совершать

```
    def attack(self, other_hero):
```

```
other_hero.hp -= self.damage
print(f"□ {self.name} атаковал {other_hero.name} на
{self.damage} урона!")
print(f"♥ У {other_hero.name} осталось {other_hero.hp} HP.")
```

Создаем двух РЕАЛЬНЫХ персонажей (объектов) по одному чертежу

```
warrior = Hero("Магнус", 150, 20)
mage = Hero("Гэндальф", 80, 35)
```

Заставляем их взаимодействовать

```
warrior.attack(mage)
```

Выгода: Нам не нужно для каждого персонажа писать отдельные функции. Мы создали один шаблон **Hero** и можем наплодить хоть 1000 разных монстров и рыцарей одной строчкой кода.

Генераторы списков (List Comprehensions)

Это способ создавать и фильтровать списки в одну строку. Вместо того чтобы создавать пустой список, писать цикл **for**, условие **if** и делать **.append()**, вы пишете всё внутри квадратных скобок.

- **Пример:** У нас есть список цен, нужно сделать список цен со скидкой 10%, но только для товаров дороже 100 рублей

Старый способ (как мы делали в базе):

```
prices = [50, 120, 250, 80, 400]
discount_prices = []
```

```
for p in prices:
    if p > 100:
        discount_prices.append(p * 0.9)
```

Новый продвинутый способ (Генератор):

```
prices = [50, 120, 250, 80, 400]
```

```
# Формула: [ЧТО ДЕЛАЕМ | ЦИКЛ FOR | УСЛОВИЕ IF]
discount_prices = [p * 0.9 for p in prices if p > 100]
```

```
print(discount_prices) # Выведет: [108.0, 225.0, 360.0]
```

Выгода: Код пишется мгновенно, занимает одну строку и работает на уровне самого интерпретатора Python быстрее, чем обычный цикл.

Лямбда-функции (`lambda`)

Это короткие анонимные функции (у них нет имени `def`), которые пишутся в одну строку. Они нужны для простых быстрых действий, которые требуются «здесь и сейчас» и больше нигде в программе не повторяются.

- **Пример: Быстрая сортировка сложных данных**

У нас есть список товаров в интернет-магазине (в виде словарей). Нам нужно отсортировать их **по цене**.

```
products = [  
    {"name": "Мышка", "price": 1500},  
    {"name": "Клавиатура", "price": 3500},  
    {"name": "Коврик", "price": 500}  
]
```

Функция `sorted` просит указать "ключ" для сортировки.

Мы передаем лямбду: `x` - это элемент списка (словарь), `x["price"]` - по чему сортировать.

```
sorted_products = sorted(products, key=lambda x: x["price"])
```

```
print(sorted_products)
```

Выведет список, отсортированный от дешевого к дорогому: Коврик -> Мышка -> Клавиатура

Выгода: Нам не пришлось создавать отдельную функцию `def get_price(x): return x['price']`, сохраняя её и передавать. Мы написали микро-инструкцию прямо внутри сортировщика.

Декораторы (`@decorator`)

Декоратор — это функция, которая принимает другую функцию, «оборачивает» её в дополнительный код и возвращает обратно.

Представьте чехол для телефона: телефон выполняет свои функции, но чехол добавляет ему водонепроницаемость и защиту.

- **Пример: Замер скорости работы функции**

Вам нужно узнать, сколько миллисекунд занимает сложный расчет в вашей программе.

```
import time
```

```
# Создаем декоратор
```

```
def timer_decorator(func):
    def wrapper():
        start_time = time.time() # Фиксируем время ДО запуска
        func()                   # Запускаем нашу основную функцию
        end_time = time.time()   # Фиксируем время ПОСЛЕ запуска
        print(f"□ Функция выполнялась за {end_time - start_time:.4f}
сек.")
    return wrapper
```

Применяем декоратор с помощью значка @

@timer_decorator

```
def heavy_calculation():
```

Имитируем долгий цикл

```
total = sum(range(10000000))
```

```
print(f"Расчет окончен. Сумма: {total}")
```

Запускаем функцию как обычно

```
heavy_calculation()
```

Выгода: Мы можем повесить **@timer_decorator** над абсолютно любой функцией в нашей программе (загрузка файлов, отправка писем, рендер карты), и она автоматически начнет замерять своё время без переписывания её внутреннего кода.

Работа с форматом **JSON** (Язык общения в интернете)

JSON (JavaScript Object Notation) — это общепринятый текстовый формат, в котором программы в интернете передают друг другу данные. По сути, **JSON** выглядит один в один как **словарь в Python**, но в виде обычной текстовой строки.

Когда ваша программа получает данные из интернета, это строка (**JSON**). Чтобы с ней работать, её нужно превратить в словарь Python (**десериализация**). А чтобы отправить данные на сайт — превратить словарь в строку (**сериализация**).

- **Пример: Превращаем JSON в словарь и обратно**

```
import json
```

Имитируем, что мы скачали данные о пользователе из интернета (это просто текст/строка)

```
json_data = '{"user_id": 452, "name": "Анна", "status": "online"}'
```

1. ЧТЕНИЕ: Метод `json.loads()` превращает текстовую строку в удобный словарь Python

```
user_dict = json.loads(json_data)
```

```
print(user_dict["name"]) # Выведет: Анна (работаем как с обычным словарем!)
```

2. ЗАПИСЬ: Изменяем данные и хотим отправить их обратно в интернет

```
user_dict["status"] = "offline"
```

Метод json.dumps() превращает словарь обратно в текстовую строку JSON

```
updated_json = json.dumps(user_dict, ensure_ascii=False)
```

```
print(updated_json) # Выведет строку: {"user_id": 452, "name": "Анна", "status": "offline"}
```

Сетевые запросы: библиотека **requests** и **API**

API (Application Programming Interface) — это «окно выдачи», которое крупные сайты открывают для программистов. Например, у банка есть **API**, которое отдает курс валют, а у погодного сайта — **API** с температурой.

С помощью библиотеки **requests** мы можем отправлять запросы к этим **API** прямо из кода Python.

- **Пример: Получаем реальный курс валют или погоду**
(Для работы нужно установить библиотеку через консоль: **pip install requests**)

```
import requests
```

Отправляем GET-запрос на публичное API (например, сайт, который определяет ваш IP и город)

```
response = requests.get("https://ipapi.co")
```

Если сайт ответил успешно (код 200)

```
if response.status_code == 200:
```

 # Метод .json() автоматически соединяет тему JSON и requests, превращая ответ в словарь!

```
    data = response.json()
```

```
    city = data.get("city", "Неизвестно")
```

```
    country = data.get("country_name", "Неизвестно")
```

```
    print(f"□ Программа определила, что вы находитесь в: {city}, {country}")
```

```
else:
```

```
    print("Не удалось связаться с сервером.")
```

Базы данных: библиотека `sqlite3` и язык `SQL`

Ранее мы сохраняли данные в текстовые файлы `.txt`. Но представьте банк или маркетплейс с миллионом клиентов. Если записать их в файл, то для поиска одного человека программе придется перечитывать терабайты текста. Это невероятно медленно. Для этого используют **Базы Данных (БД)**. Данные там хранятся в строгих таблицах, а встроенный язык `SQL` позволяет находить нужную строчку среди миллиардов за миллисекунды. В Python уже «встроена» легкая база данных — `SQLite`.

- **Пример: Создаем базу данных пользователей и ищем игрока**

```
import sqlite3
```

```
# 1. Подключаемся к файлу базы данных (если его нет, он создастся сам)
```

```
connection = sqlite3.connect("game_database.db")
```

```
cursor = connection.cursor()
```

```
# 2. Создаем таблицу игроков (если она еще не создана)
```

```
# Используем язык SQL внутри Python-строки:
```

```
cursor.execute("""
```

```
CREATE TABLE IF NOT EXISTS players (
```

```
    id INTEGER PRIMARY KEY,
```

```
    nickname TEXT,
```

```
    level INTEGER
```

```
)
```

```
""")
```

```
# 3. Добавляем нового игрока в таблицу
```

```
cursor.execute("INSERT INTO players (nickname, level) VALUES ('CyberKnight', 42)")
```

```
connection.commit() # Обязательно сохраняем изменения в базе
```

4. БЫСТРЫЙ ПОИСК: Вытаскиваем игрока с уровнем больше 40

```
cursor.execute("SELECT nickname FROM players WHERE level > 40")
```

```
high_level_players = cursor.fetchall() # Забираем результаты поиска
```

```
print("Игроки высокого уровня:")
```

```
for player in high_level_players:
```

```
    print(f"- {player[0]}") # Выведет: - CyberKnight
```

```
# Закрываем соединение
```

```
connection.close()
```

Как это работает в связке? (Реальный сценарий)

Вы пишете **программу-агрегатор** авиабилетов:

1. Ваша программа делает сетевой запрос через **requests** на сайт Аэрофлота.
2. Сервер Аэрофлота возвращает ответ в формате **JSON**.
3. Python превращает **JSON** в словарь, вытаскивает оттуда самые дешевые билеты и мгновенно сохраняет их в **Базу Данных (SQL)**, чтобы затем показать пользователю в красивом интерфейсе.

Асинхронное программирование (**asyncio, async/await**)

Обычный код в Python выполняется **синхронно** (строго по очереди). Если ваша программа делает сетевой запрос к сайту, который долго отвечает, вся программа полностью «зависает» на эти 5 секунд. Она не может реагировать на нажатия кнопок или обслуживать других пользователей.

Асинхронность решает эту проблему. Программа становится похожа на опытного повара в ресторане: пока закипает вода для пасты (идет

запрос в интернет), он не стоит на месте, а режет салат для другого заказа.

Для этого используются ключевые слова **async** (обозначает асинхронную функцию) и **await** (команда «подожди, пока выполнится это долгое дело, но не замораживай остальной код»).

- **Пример: Имитируем одновременную загрузку трех файлов без зависания**

```
import asyncio
import time
```

```
# async def делает функцию асинхронной
async def download_file(file_name, seconds):
    print(f"Началась загрузка файла: {file_name}")
    # Вместо time.sleep(), который намертво вешает программу,
    # мы используем асинхронное ожидание asyncio.sleep()
    await asyncio.sleep(seconds)
    print(f"✔️ Файл {file_name} успешно скачан за {seconds} сек.!")
```

```
async def main():
    start = time.time()

    # Запускаем три скачивания одновременно (параллельно)
    await asyncio.gather(
        download_file("База_данных.zip", 3),
        download_file("Картинка.png", 1),
        download_file("Песня.mp3", 2)
    )

    end = time.time()
    print(f"☐ Все файлы скачаны! Общее время работы: {end - start:.2f} сек.")
```

```
# Запуск асинхронного движка
asyncio.run(main())
```

Магия результата: Синхронная программа скачивала бы их по очереди и потратила бы $(3 + 1 + 2 = 6)$ секунд. Асинхронная программа запустит их вместе и выполнит всё за **3 секунды** (по времени самого долгого файла)! Именно на этом движке пишутся современные быстрые Telegram-боты на библиотеке **aiogram**.

Управление пакетами и Виртуальные окружения (**pip**, **venv**)

Когда вы начнете создавать реальные проекты (например, сайты или нейросети), вы будете скачивать много чужих библиотек с помощью команды **pip install** (библиотеки для работы с графикой, серверами, базами данных).

Но представьте: для Проекта №1 нужна старая версия библиотеки **Django 3.2**, а для нового Проекта №2 нужна свежая **Django 5.0**. Если ставить их прямо в компьютер, они начнут конфликтовать, перезаписывать друг друга, и ваши программы просто сломаются.

Виртуальное окружение (venv) создаёт изолированную, чистую «коробку» для каждого вашего отдельного проекта. Внутри этой коробки находится своя копия Python и свой набор библиотек, который никак не пересекается с другими проектами.

Как это выглядит на практике (Команды в терминале):

Создаем изолированную коробку в папке проекта:

```
python -m venv my_env
```

В проекте появится папка **my_env** со своим изолированным Python).

2. Активируем окружение (заходим внутрь коробки):

- Для Windows: **my_env\Scripts\activate**
 - Для Mac/Linux: **source my_env/bin/activate**
- (В начале строки терминала появится плашка (**my_env**), сигнализирующая, что вы внутри).

3. Устанавливаем нужные библиотеки:

```
pip install requests
```

1. Теперь библиотека **requests** установилась только в этот проект. Стоит закрыть проект или удалить папку **my_env** — компьютер снова останется абсолютно чистым.

Алгоритм создания программы

ЭТАП 1: Анализ и Проектирование (О чем думать ДО кода)

Главная цель этапа — понять, ЧТО мы делаем, ЗАЧЕМ и КАК это будет работать в теории.

- Шаг 1. Формулируем идею и ценность

- *Вопрос себе: Какую конкретную задачу решает моя программа?*
- *Пример: «Я хочу написать Telegram-бота для учета личных расходов, чтобы видеть, куда уходят деньги».*
- **Шаг 2. Составляем ТЗ (Техническое задание) для себя**
 - Выпишите списком минимальный набор функций (MVP — минимально жизнеспособный продукт), без которых программа бессмысленна. Не пытайтесь сделать всё и сразу.
 - *Пример: Бот должен: 1) Принимать сумму; 2) Спрашивать категорию (еда, аренда, авто); 3) Сохранять это; 4) По команде выдавать общую сумму за месяц.*
- **Шаг 3. Проектируем архитектуру данных (Выбор инструментов)**
 - Подумайте, какие типы данных и инструменты из выученных уровней вам понадобятся.
 - *Где хранить настройки? (В файле или словаре **dict**).*
 - *Нужна ли база данных? (Если пользователей много — нужна **SQLite3**).*
 - *Нужна ли сеть? (Да, библиотека **requests** или **aiogram** для связи с серверами Telegram).*

ЭТАП 2: Подготовка рабочего окружения (Старт проекта)

Главная цель этапа — создать чистую и изолированную среду на компьютере.

- **Шаг 4. Создаем структуру папок**
 - Создайте на компьютере новую папку с понятным названием проекта (например, **money_tracker_bot**).
- **Шаг 5. Включаем Виртуальное окружение (venv)**
 - *Зачем:* Чтобы библиотеки этого проекта не конфликтовали с другими.
 - *Действие:* Открываем терминал в папке проекта и пишем: **python -m venv venv**. Активируем его.
- **Шаг 6. Устанавливаем внешние библиотеки (pip)**
 - Устанавливаем только то, что нужно для минимальной версии (например, **pip install aiogram**).
- **Шаг 7. Создаем файлы**
 - Создаем главный файл **main.py** (наша точка входа) и, если нужно, вспомогательные файлы (например, **database.py** для работы с БД).

Написание кода (Пошаговая сборка)

Главная цель этапа — двигаться от простого к сложному, проверяя каждый шаг.

- **Шаг 8. Прописываем «скелет» и константы**

- В самом верху файлов делаем импорты (`import os, asyncio...`).
- Записываем неизменяемые правила игры в **Кортежи** (`tuple`) или глобальные переменные большими буквами (например, `TOKEN = "12345", CATEGORIES = ("Еда", "Транспорт", "Развлечения")`).

- **Шаг 9. Пишем логику в виде чистых Функций (def)**

- Разбивайте код на маленькие функции, где каждая отвечает строго за ОДНО действие.
- *Пример:* Отдельная функция `connect_db()` (только подключается к базе), отдельная функция `add_expense(amount, category)` (только записывает данные).

- **Шаг 10. Добавляем защиту и безопасность (try / except)**

- Оборачивайте опасные места (ввод пользователя, деление, чтение файлов, сетевые запросы) в блоки `try/except`. Программа не должна падать, если пользователь случайно введет буквы вместо цифр.

- **Шаг 11. Оформляем точку входа**

- В самом низу файла `main.py` жестко прописываем конструкцию:

```
if __name__ == '__main__':
```

```
# Здесь запускаем главный цикл программы или бота
```

ЭТАП 4: Тестирование и Полировка (Доведение до ума)

Главная цель этапа — сломать свою программу раньше, чем это сделает пользователь.

- **Шаг 12. "Тест дурака"**

- Запустите программу и попробуйте вводить неверные данные, нажимать не те кнопки, отключать интернет. Смотрите, как ведет себя код. Если вылетает ошибка — возвращаемся к Шагу 10 и добавляем `except`.

- **Шаг 13. Рефакторинг (Чистка кода)**

- Уберите лишние `print()`, которые вы использовали для тестов.
- Посмотрите на длинные циклы — возможно, их стоит заменить на быстрые и красивые **Генераторы списков** (`List Comprehensions`).

Золотые правила разработчика (Запишите на полях):

1. **Принцип KISS (Keep It Simple, Stupid):** Делайте код максимально простым. Если алгоритм можно написать без «магии» и наворотов — пишите просто. Его потом будет легче читать вам же.
2. **Не пишите код дольше 30 минут без запуска:** Написали одну функцию — запустили и проверили через `print()`. Нашли ошибку сразу — исправили. Если написать 200 строк кода и запустить только в конце, искать баг придется часами.

Развилка А: Линейно-функциональный подход (Скрипты и Утилиты)

- **Суть:** Программа работает сверху вниз. Она делает вычисления, автоматизирует рутину или обрабатывает файлы.
- **Главный инструмент Python:** Функции `def`, модули `os`, чтение/запись файлов.
- **Примеры программ:** Робот для очистки папок от старых файлов, парсер сайтов, конвертер валют, калькулятор скидок.
- **О чём думать на Этапе 1:**
 - Каковы входные данные (файл, текст, число)?
 - Каков чёткий пошаговый алгоритм их обработки?
 - Куда сохранить результат (в новый файл или вывести на экран)?

Развилка Б: Событийно-ориентированный подход (Интерактивные программы)

- **Суть:** Программа не выполняет код сама по себе. Она **ждёт** действия от внешнего мира (клик мышкой, сообщение от пользователя, сигнал таймера) и реагирует на него.
- **Главный инструмент Python:** Асинхронность (`async/await`), библиотеки `aiogram` (для ботов) или `FastAPI` (для сайтов).
- **Примеры программ:** Telegram-боты, веб-сайты, чаты, любые приложения с кнопками.
- **О чём думать на Этапе 1:**
 - Каковы сценарии поведения пользователя (что если он нажмёт кнопку А, а что если кнопку Б)?
 - Как защитить программу от одновременных запросов сотен людей (асинхронность)?

Развилка В: Объектно-ориентированный подход (Сложные системы)

- **Суть:** Программа моделирует реальный мир. Мы создаём виртуальные объекты, которые обладают свойствами и могут взаимодействовать друг с другом.
- **Главный инструмент Python:** Классы (`class`), **ООП**.

- **Примеры программ:** Игры (герои, монстры, предметы), CRM-системы (клиенты, заказы, сотрудники).
- **О чём думать на Этапе 1:**
 - Какие «сущности» есть в моей программе?
 - Какими характеристиками они обладают (свойства) и что они умеют делать (методы)?

Если на Этапе 1 стоит задача...	...то в коде я 100% использую:	Почему именно это?
Зафиксировать жесткие правила (настройки, цвета, ID каналов)	Кортеж (tuple)	Никакой баг в коде не сможет случайно их перезаписать.
Сделать изменяемый список (корзина товаров, инвентарь)	Список (list)	Элементы можно легко добавлять, удалять и сортировать.
Убрать повторения (уникальные IP посетителей, чистые Email)	Множество (set)	Python сам мгновенно очистит дубликаты без циклов.
Создать подробную анкету (карточка товара, профиль юзера)	Словарь (dict)	Данные лежат под понятными текстовыми ключами, а не номерами.
Связать программу с интернетом (погода, курсы валют)	Библиотека requests + JSON	JSON — это мировой стандарт общения сайтов.
Сохранить миллионы строк (история операций, база клиентов)	База данных SQL / sqlite3	Поиск по файлу .txt зависнет, а SQL найдет строку за миллисекунду.

Запомните правило: **Никогда не учите код абстрактно.**

Если вы открываете тему «Словари», сразу задавайте себе вопрос из Этапа 1: *«В какой программе мне это пригодится? Ага, сделаю телефонную книгу»*. Если проходите «Асинхронность»: *«Так, это спасёт моего Telegram-бота от зависания, когда ему пишут 10 человек одновременно»*.

ПРИМЕР: Наша задача на **Этапе 1**: создать скрипт (Развилка А — линейная утилита), который отправит сигналы (ping-запросы) на все IP-адреса в вашей домашней Wi-Fi сети, чтобы узнать, какие устройства (телефоны, умные лампы, ноутбуки) сейчас включены.

Ниже представлен пошаговый разбор кода для вашего конспекта — строка за строкой, с объяснением **зачем** пишется каждый элемент.

Создание Wi-Fi Ping-сканера

Шаг 1. Импортируем инструменты (Подготовка)

Программу мы начинаем не с логики, а с подключения «помощников». Нам нужно общаться с операционной системой.

import os # Импортируем встроенный модуль OS. Он нужен, чтобы Python мог давать команды вашей операционной системе (Windows/Mac).

import platform # Импортируем модуль platform. Он нужен, чтобы программа сама поняла, где она запущена (на Windows или Linux/Mac), так как команды отправки сигналов там немного различаются.

Шаг 2. Создаем «Скелет» и универсальную функцию сигналов

Теперь пишем функцию **def**, которая будет отправлять один сигнал (**ping**) на конкретный **IP-адрес**.

def ping_device(ip_address):

Зачем эта функция: она принимает один IP-адрес, шлет ему запрос и говорит, ответил он или нет.

1. Развилка под ОС. Windows использует флаг "-n" для указания количества запросов, а Mac/Linux используют флаг "-c".

current_os = platform.system().lower()

if "windows" in current_os:

command_flag = "-n"

else:

command_flag = "-c"

2. Формируем f-строку команды для терминала.

Мы говорим операционной системе: "отправь ровно 1 сигнал на этот IP".

Добавляем "> nul" или "> /dev/null", чтобы скрыть системный мусор терминала с экрана.

if "windows" in current_os:

command = f"ping {command_flag} 1 {ip_address} > nul"

else:

command = f"ping {command_flag} 1 {ip_address} > /dev/null 2>&1"

Шаг 3. Добавляем безопасность и отправку (**try / except**)

Вставляем «подушку безопасности» прямо в момент отправки сигнала. Вдруг сеть заблокирована или адрес некорректен

try:

Функция `os.system()` берет нашу `f`-строку и выполняет ее в командной строке компьютера.

Если устройство в сети и ответило, операционная система вернет код 0.

response = os.system(command)

Проверяем ответ системы

if response == 0:

return True # Устройство онлайн, сигнал вернулся!

else:

return False # Устройство выключено или скрыто.

except Exception as error:

Если произойдет сбой (например, отключилась сетевая карта), программа не упадет.

print(f"Ошибка при пинге адреса {ip_address}: {error}")

return False

Шаг 4. Пишем цикл перебора сети и точку входа

Теперь объединяем функцию с циклом **for**, чтобы проверить целый диапазон адресов (обычно домашние роутеры раздают адреса от **192.168.1.1** до **192.168.1.254**).

if __name__ == '__main__':

Наша точка входа. Защищает код от случайного автозапуска при импорте.

print("□ Запуск сканирования Wi-Fi сети... Пожалуйста, подождите.")

Базовая часть домашнего IP-адреса

base_ip = "192.168.1."

Создаем изменяемый список (list), куда будем сохранять найденные устройства.

```
active_devices = []
```

Запускаем цикл for для перебора чисел от 1 до 20 (для экономии времени теста, проверим первые 20 адресов).

```
for host in range(1, 21):
```

Собираем полный IP-адрес: соединяем "192.168.1." и текущее число цикла (host)

```
current_ip = f"{base_ip}{host}"
```

Вызываем нашу функцию, которую написали на Шаге 2

```
is_online = ping_device(current_ip)
```

```
if is_online:
```

```
    print(f"□ Найдено устройство: {current_ip}")
```

```
    active_devices.append(current_ip) # Добавляем в список успехов
```

```
else:
```

Чтобы в консоли было видно движение, пишем серую точку для выключенных IP

```
    print(f"● {current_ip} - пустой адрес", end="\r")
```

Финал: Выводим итоги

```
print("\n--- Сканирование завершено! ---")
```

```
print(f"Всего активных устройств в вашей Wi-Fi сети: {len(active_devices)}")
```

```
print(f"Список их IP: {active_devices}")
```

скрипт задействовал **все уровни**, которые вы изучили:

1. **База:** Условия **if/else** определили операционную систему, а цикл **for** перебрал числовой диапазон **range()**.
2. **Списки:** Мы создали пустой рюкзак **active_devices = []** и наполняли его методом **.append()**.
3. **Модули и файлы:** Мы использовали **import os** для прямого контроля железа компьютера.
4. **Строки:** Мы динамически собирали адреса через **f"{base_ip}{host}"**.
5. **Безопасность:** Блок **try/except** защитил скрипт от падения при сбоях сети.
6. **Архитектура:** Код упакован в функцию **def** и запускается строго через **if __name__ == '__main__':**.
Если вы примените этот код в **IDE** (например, **PyCharm** или **VS Code**) и запустите, ваш компьютер начнет посылать реальные микро-сигналы в роутер и покажет, кто сейчас подключен к вашему Wi-Fi.

Модификация этого сканера с помощью асинхронности — это лучший способ увидеть разницу между «просто кодом» и «высокопроизводительным кодом»

Асинхронный Wi-Fi сканер с функцией-оберткой

Шаг 1. Импорт асинхронного движка

К стандартным модулям мы добавляем главного виновника торжества — **asyncio** и модуль **time** для замера скорости.

```
import os
```

```
import platform
```

```
import asyncio # Импортируем asyncio — библиотеку, которая  
позволяет функциям не ждать друг друга, а выполняться  
параллельно.
```

```
import time # Импортируем time, чтобы обертка могла фиксировать  
точное время на часах компьютера.
```

Шаг 2. Создаем Декоратор и Функцию-Обертку (**wrapper**)

Декоратор — это функция, которая принимает другую функцию как коробку, создает вокруг нее «обертку», добавляет новые действия и возвращает обратно.

```
def timer_decorator(func):
```

```
    # Это сам декоратор. Он принимает функцию 'func' (наш сканер).
```

async def wrapper(*args, **kwargs):

А вот это та самая ОБЕРТКА (wrapper). Она «оборачивает» функцию.

Так как наш сканер асинхронный, обертка тоже должна быть async.

start_time = time.time() # 1. Действие ДО: Засекаем время старта на часах

result = await func(*args, **kwargs) # 2. Запускаем саму основную функцию (сканер) и ждем ее финиша через await

end_time = time.time() # 3. Действие ПОСЛЕ: Засекаем время окончания

print(f"□ ВРЕМЯ РАБОТЫ ОБЕРТКИ: Сканирование заняло всего {end_time - start_time:.2f} сек.!")

return result # Возвращаем результат работы основной функции

return wrapper # Декоратор отдает готовую "обернутую" функцию обратно в программу

Шаг 3. Асинхронная функция для ОДНОГО сигнала

Превращаем обычный пинг в асинхронный. Обратите внимание на ключевые слова.

async def ping_device_async(ip_address):

Слово 'async' перед def означает: "Эта функция может встать на паузу, уступив место другим".

current_os = platform.system().lower()

command_flag = "-n" if "windows" in current_os else "-c"

if "windows" in current_os:

command = f"ping {command_flag} 1 -w 1000 {ip_address} > nul"

Добавили -w 1000 (таймаут 1 секунда, чтобы Windows не ждала ответа вечно)

else:

command = f"ping {command_flag} 1 -W 1 {ip_address} > /dev/null 2>&1" # Добавили -W 1 (таймаут 1 секунда для Linux/Mac)

try:

ОС не умеет работать асинхронно сама по себе.

```

# Поэтому мы вызываем специальный инструмент
asyncio.to_thread, который запускает команду os.system в отдельном
фоновом потоке, не мешая основному коду.
response = await asyncio.to_thread(os.system, command)

if response == 0:
    return ip_address # Если устройство онлайн, возвращаем его
IP
return None

except Exception:
    return None

```

Шаг 4. Главный асинхронный менеджер и применение ОБЕРТКИ

Вот тут мы применяем нашу обертку с помощью значка @.

```

@timer_decorator # Магия! Мы "обернули" функцию main в наш
таймер. Теперь при ее запуске автоматически сработает код из Шага
2.
async def main():
    base_ip = "192.168.1."

    # Создаем список "задач" для Python.
    # С помощью Генератора списков (тема Уровня 1) мы мгновенно
    создаем 20 асинхронных задач на пинг.
    tasks = [ping_device_async(f"{base_ip}{host}") for host in range(1,
21)]

    print(f"□ Запуск параллельного сканирования {len(tasks)}
адресов...")

    # МЕГА-КОМАНДА: asyncio.gather запускает ВСЕ 20 задач
ОДНОВРЕМЕННО.
    # Программа выстреливает 20 сигналами в сеть в одну и ту же
миллисекунду!
    results = await asyncio.gather(*tasks)

    # Очищаем результаты от пустых ответов None (оставляем только
живые IP)
    active_devices = [ip for ip in results if ip is not None]

    print("\n--- Результаты ---")
    print(f"Найдено живых устройств: {len(active_devices)}")

```

```
print(f"Их адреса: {active_devices}")
```

Шаг 5. Запуск через асинхронный движок

```
if __name__ == '__main__':
```

```
# Асинхронные функции нельзя запустить просто как main().
```

```
# Нужен специальный стартер - asyncio.run(), который запускает  
внутренний цикл событий (Event Loop).
```

```
asyncio.run(main())
```

В программировании **порядок слов и шагов в декораторах жестко продиктован логикой интерпретатора Python**. Если нарушить этот порядок или пропустить хотя бы одно ключевое слово, код либо мгновенно вылетит с ошибкой, либо начнет вести себя непредсказуемо.

Разберем как пример строку кода в плане для чего и зачем:

```
async def wrapper(*args, **kwargs):
```

Эта строка — универсальный «входной шлюз» обертки. Давайте разберем каждое слово слева направо:

1. **async** (Асинхронность)

- **Зачем:** Указывает Python, что код внутри обертки имеет право использовать команду **await** (то есть вставлять на паузу и пропускать другие задачи вперед).

- **Что будет, если убрать:** Программа выдаст ошибку **SyntaxError** на строке с **await func()**. Python запрещает использовать **await** внутри обычных (синхронных) функций.

2. **def** (Определение)

- **Зачем:** Стандартное системное слово Python, заявляющее: «Внимание, прямо сейчас я создаю новую функцию».

3. **wrapper** (Имя)

- **Зачем:** Это просто техническое название переменной-функции. Вы можете назвать её как угодно, например **def obertka**, но в мировом сообществе программистов принято использовать слово **wrapper** (обертка), чтобы другие разработчики сразу понимали архитектуру кода.

4. ***args** (Все позиционные аргументы)

- **Зачем:** Звездочка ловит абсолютно любые аргументы, переданные через запятую (например: 5, "текст", True), и пакует их в кортеж.
- 5. ****kwargs** (Все именованные аргументы — **Keyword Arguments**)
 - **Зачем:** Две звездочки ловят аргументы, которые передаются по именам (например: timeout=1, mode="fast"), и пакуют их в словарь **dict**.
 - **Почему они стоят вместе (*args, **kwargs):** Это универсальная «ловушка». Декоратор не знает заранее, какую именно функцию вы решите им обернуть — функцию без аргументов, функцию с тремя числами или функцию со словарем. Эта комбинация говорит: «Я приму в себя абсолютно любые данные, в любом количестве, какими бы они ни были».
 - **Что будет, если убрать их и оставить пустые скобки ():** Если вы попытаетесь применить этот декоратор к функции, которая принимает хоть какие-то данные (например, **ping_device(ip_address)**), программа мгновенно упадет с ошибкой **TypeError: wrapper() takes 0 positional arguments but 1 was given**. Обертка просто «задохнется», не сумев принять входящие данные.

последовательность шагов внутри обертки

Внутри функции **wrapper** логика строится строго по схеме: «До» ➔ «Событие» ➔ «После» ➔ «Возврат».

Давайте посмотрим, почему нельзя менять их местами:

Шаг 1: start_time = time.time() (Фиксация ДО)

- **Зачем:** Мы включаем секундомер в самую первую микросекунду, как только управление перешло в обертку.
- **Что будет, если пропустить:** Мы потеряем точку отсчета. Позже мы не сможем вычислить, сколько времени прошло.

Шаг 2: result = await func(*args, **kwargs) (Запуск цели)

- **Зачем:**
 - Слово **await** здесь критически важно: оно приказывает программе: «Запусти основную функцию **func** и дождись её полного завершения, но пока ты ждешь, не зависай, а дай поработать другим фоновым задачам».
 - Переменная **result** захватывает то, что вернет функция (в нашем случае — список живых IP-адресов).
 - В скобки (***args, **kwargs**) мы распаковываем те самые данные, которые поймали в заголовке строки. Мы передаем их дальше «оригинальной» функции в целости и сохранности.
- **Что будет, если поменять местами Шаг 1 и Шаг 2:** Если сначала запустить функцию, а только потом включить секундомер

`start_time`, то время выполнения функции вообще не запишется. Секундомер включится уже *после* того, как сканер завершит работу.

- **Что будет, если забыть слово `await`:** Программа не запустит сканер. Вместо этого в переменную `result` запишется технический объект «корутина» (обещание выполнения), и код пойдет дальше, выдав пустой результат за 0 секунд.

Шаг 3: `end_time = time.time()` (Фиксация ПОСЛЕ)

- **Зачем:** Программа выключает секундомер сразу после того, как `await func()` отмерла и выдала результат. Математика `end_time - start_time` дает нам чистую скорость работы.
- **Что будет, если поставить эту строку ДО Шага 2:** Мы замерим скорость работы самого процессора по включению секундомера, а не работу сканера. Разница во времени будет равна нулю.

Шаг 4: `return result` (Финальный возврат)

- **Зачем:** Обертка обязана отдать наружу (тому коду, который её вызвал) результат работы оригинальной функции.
- **Что будет, если пропустить этот шаг:** Функция отработает, найдет устройства в сети, но наружу ничего не отдаст. В Python функция, у которой нет слова `return`, автоматически возвращает пустоту — `None`. Ваша основная программа в итоге подумает, что сканер ничего не нашел.

`return wrapper` (в самом низу декоратора)

Обратите внимание на отступ этой строки — она находится **вне** функции `wrapper`, но **внутри** функции `timer_decorator`.

- **Зачем:** Когда вы пишете над функцией значок `@timer_decorator`, Python берет вашу оригинальную функцию, закидывает ее в декоратор и ждет, что декоратор отдаст взамен модифицированную версию. Строка `return wrapper` как раз и отдает эту готовую «одетую в таймер» обертку обратно в систему.
- **Что будет, если забыть эту строку:** Декоратор вернет пустоту (`None`). В этот же момент ваша оригинальная функция `main` навсегда сотрется из памяти компьютера, превратившись в `None`. При попытке её запуска программа выдаст ошибку `TypeError: 'NoneType' object is not callable`.

ШАГ 1: Самые первые строки в файле — это всегда импорты. Мы не можем использовать продвинутые инструменты, пока не заявим о них компьютеру.

- **Строка 1: `import os`** (Даем программе доступ к командам операционной системы).

- **Строка 2: `import asyncio`** (Включаем асинхронный движок, если программа должна делать несколько дел параллельно).

ШАГ 2: Строки правил и законов («В каких условиях работаем?»)

Сразу под импортами мы создаем глобальные настройки. Они должны быть объявлены до того, как их начнут читать функции ниже.

- **Строка 3: `CATEGORIES = ("Еда", "Спорт")`** (Записываем неизменяемый кортеж-правило).
- **Строка 4: `BASE_IP = "192.168.1."`** (Задаем стартовый текст для адресов).

ШАГ 3: Строки создания инструментов («Записываем рецепты»)

Далее идут блоки функций `def`. Мы учим компьютер новым командам. Код внутри этих строк пока не запускается, он просто сохраняется в память компьютера как инструкция.

- **Строка 5: `def timer_decorator(func):`** (Создаем функцию-декоратор).
- **Строка 6: `async def wrapper(*args):`** (Внутри декоратора первой строкой открываем функцию-обертку, которая перехватит управление).
- **Строка 7: `start_time = time.time()`** (Первой строкой внутри обертки включаем секундомер).
- **Строка 8: `result = await func(*args)`** (Следующей строкой запускаем саму задачу и ждем её).
- **Строка 9: `return result`** (Финальной строкой обертки отдаем результат наружу).
- **Строка 10: `async def main():`** (Создаем главную функцию-управленца).

ШАГ 4: Строки запуска («Поехали!»)

Самый низ файла — это кнопка «СТАРТ». Дойдя до этих строк, Python понимает, что все инструкции выше прочитаны, и пора начинать работу.

- **Строка 11: `if __name__ == '__main__':`** (Проверяем, что файл запущен напрямую).
- **Строка 12: `asyncio.run(main())`** (Следующей строкой включаем главный асинхронный движок и передаем ему управление).

Что будет, если нарушить этот построчный порядок? (Примеры катастроф)

1. Если поставить ШАГ 4 (Запуск) в самое начало файла: Программа мгновенно упадет с ошибкой `NameError: name 'main' is not defined`. Python читает код сверху вниз. Если вы прикажете

ему запустить `main()` до того, как написали строку `def main():`, он ответит: «Я понятия не имею, что это за команда, вы мне о ней еще не рассказывали».

2. Если поставить ШАГ 1 (Импорт) ниже ШАГа 3:

Как только Python дойдет до строки с использованием встроенного модуля (например, `os.system()`), он сломается, так как строка `import os` еще не была прочитана и модуль не подключен.

В Python сначала нужно **привезти инструменты (импорт)**, затем **установить правила (константы)**, после этого **написать инструкции (функции/классы)** и только в самом конце **нажать на кнопку запуска (main)**

Как читать чужой код и искать багги

ЧАСТЬ 1: Алгоритм разбора чужого кода («С чего начать?»)

Когда перед вами открывается огромный файл с незнакомым кодом, не пытайтесь читать его как художественную книгу с первой строки. Действуйте по шагам:

1. **Найдите точку входа (`if __name__ == '__main__':`)**
 - Зачем: Это сердце программы. Спуститесь в самый низ файла и посмотрите, какая функция запускается первой (обычно это `main()`).
2. **Двигайтесь снизу вверх по цепочке вызовов**
 - Посмотрите, что делает главная функция. Она вызывает функцию `connect_db()` и `start_bot()` Об? Перейдите к их определениям выше по коду. Так вы поймете маршрут, по которому ходят данные.
3. **Изучите глобальные константы вверху файла**
 - Словари характеристик, кортежи с настройками и пути к файлам сразу объяснят вам «правила игры» этой программы.
4. **Разделяйте код на смысловые блоки**
 - Мысленно (или комментариями) сворачивайте функции. Ваша цель — понять не каждую строчку, а общую картину: «Этот блок отвечает за отправку сообщений, этот — за расчет цен, а этот — за запись в лог».

ЧАСТЬ 2: Карта топ-5 распространенных багов Python

Если чужая (или ваша) программа упала с ошибкой, Python всегда выводит **Traceback** (отчет об ошибке). В самом низу этого отчета написан тип ошибки. Вот главные враги, которых нужно знать в лицо:

1. **NameError: name '...' is not defined**

- **Что значит:** Компьютер встретил слово (переменную или функцию), которую он не знает.
- **Типичная причина:**
 - Банальная опечатка в названии (написали `player_hp`, а ниже вызвали `player_hb`).
 - Нарушен **построчный порядок**: функцию попытались вызвать *выше* в коде, чем её создали через `def`.
 - **Область видимости**: попытка вызвать из основного кода локальную переменную, которая была создана внутри функции и «умерла» там.

2. **TypeError: '...' object is not ...**

- **Что значит:** Вы пытаетесь совершить действие с типом данных, который для этого не предназначен.
- **Типичная причина:**
 - Попытка сложить строку и число (`"Цена: " + 100`).

Решение: использовать f-строки.

 - Попытка изменить элемент в **кортеже** (`my_tuple[0] = 5`).

Напоминаем: кортежи неизменяемы!

 - Забыли круглые скобки при вызове функции (`my_func` вместо `my_func()`), из-за чего Python пытается работать с самой инструкцией, а не с её результатом.

3. **IndexError: list index out of range**

- **Что значит:** Вы пытаетесь достать из списка элемент по номеру (индексу), которого не существует.
- **Типичная причина:** В списке всего 3 элемента (индексы 0, 1, 2), а код пытается вызвать элемент с индексом 3 (который был бы четвертым). Помните, что отсчет всегда идет с нуля!

4. **KeyError: '...'**

- **Что значит:** Ошибка-близнец предыдущей, но для **словарей** (`dict`). Программа запросила из анкеты ключ, которого там нет.
- **Типичная причина:** Попытка вызвать `user["phone"]`, когда пользователь не заполнил это поле. *Решение:* всегда использовать метод безопасного чтения `.get()` (например, `user.get("phone", "Нет номера")`).

5. **AttributeError: 'NoneType' object has no attribute '...'**

- **Что значит:** Самая коварная ошибка новичков. Вы пытаетесь вызвать метод (например, `.append()` или `.split()`) у пустоты (`None`).
- **Типичная причина:** Функция, которая должна была вытащить данные (например, скачать из интернета или найти в базе), ничего не

нашла и вернула `None`. А следующая строка кода попыталась обработать этот `None` как нормальные данные.

ЧАСТЬ 3: Золотые правила дебаггинга (Как искать скрытые баги)

Что делать, если программа НЕ вылетает с ошибкой, но считает данные неправильно (скрытый логический баг)?

- **Правило 1: Метод «Принтования» (print-debugging)**

Расставьте по ходу движения кода временные команды `print()`. Выводите на экран состояние переменных на каждом шаге.

- *Пример:* `print(f"DEBUG: На шаге 3 переменная x равна {x}").` Так вы поймаете момент, где цифры сошли с ума.

- **Правило 2: Изолируйте проблему**

Если сломался большой асинхронный сканер или бот, скопируйте одну подозрительную функцию в отдельный пустой файл. Запустите её там в блоке `if __name__ == '__main__':` с фейковыми ручными данными. Если в маленьком файле она работает — проблема не в ней, а в том, как ей передают данные другие функции.

- **Правило 3: Читайте чужие комментарии и Docstrings**

Хорошие разработчики пишут под строкой `def` текст в тройных кавычках `"""..."""`. Это краткая инструкция, которая объясняет, какие аргументы функция ждёт на вход и что она обязана вернуть через `return`.

Анатомия **Traceback**: Учимся читать ошибки

Представьте, что мы запустили наш сканер или программу учета расходов, и в консоли выскочило вот такое страшное сообщение:

Traceback (most recent call last):

File "C:/projects/main.py", line 24, in <module>

process_user_data()

File "C:/projects/main.py", line 15, in process_user_data

calculate_discount(user_profile)

File "C:/projects/main.py", line 8, in calculate_discount

final_price = user["price"] * 0.9

KeyError: 'price'

Золотое правило чтения: Читаем СНИЗУ ВВЕРХ!

Запомните: **самая важная информация всегда находится на самой последней строчке**. Python раскручивает ошибку в обратном порядке — от финала катастрофы к ее началу.

Давайте разберем этот Traceback построчно, двигаясь снизу вверх:

Строка 1 (Самый низ — Главный диагноз)

KeyError: 'price'

- **Что это:** Это тип ошибки и ее краткое описание.
- **Как читать:** Мы уже знаем из карты багов, что **KeyError** — это ошибка **словаря**. Программа залезла в какую-то анкету (**dict**) и попыталась достать данные по ключу **'price'**, но такого заголовка внутри словаря просто не существует!

Строка 2 (Где конкретно взорвался код)

**File "C:/projects/main.py", line 8, in calculate_discount
final_price = user["price"] * 0.9**

- **Как читать:** Поднимаемся на строчку выше. Python показывает точное место аварии:
 - **File "..."** — путь к файлу, в котором всё случилось (файл **main.py**).
 - **line 8** — **номер строки**, на которой компьютер споткнулся (строка №8).
 - **in calculate_discount** — название функции, внутри которой это произошло.
 - **final_price = user["price"] * 0.9** — Python услужливо цитирует эту сломанную строку. Мы видим: ага, именно тут мы обратились к несуществующему ключу **price**.

Строки 3 и 4 (Chain of Trust — Цепочка вызовов)

**File "C:/projects/main.py", line 15, in process_user_data
calculate_discount(user_profile)**

- **Как читать:** Поднимаемся еще выше. Эта строка объясняет, **как программа вообще докатилась до жизни такой**. Она говорит: «Функция **calculate_discount**, которая сломалась на 8-й строке, была вызвана на 15-й строке внутри другой функции — **process_user_data**».
- **Зачем это нужно программисту:** Часто баг кроется не там, где код упал, а там, где функцию вызвали. Глядя на эту строку, мы

понимаем: на строке 15 в функцию передали переменную `user_profile`. Значит, в словаре `user_profile` изначально не было ключа `price`! Нужно идти туда и проверять, почему анкета пришла пустой.

Строки 5 и 6 (Самый верх — Источник запуска)

File "C:/projects/main.py", line 24, in <module>
process_user_data()

- **Как читать:** Самый верх Traceback. Слово `<module>` означает основной текст программы (наш блок `if __name__ == '__main__':`).
- **Смысл:** Всё началось на 24-й строке файла, когда мы нажали «Старт» и вызвали самую первую функцию `process_user_data()`.

Чек-лист: Что делать, если выскочил Traceback?

Когда код упал, выполните 3 быстрых шага:

1. **Посмотрите на самую нижнюю строку.** Определите тип ошибки (это `ValueError`, `TypeError`, `KeyError` или `IndexError`?).
2. **Найдите в блоке выше строку кода из ВАШЕГО файла.** (Иногда в Traceback попадают файлы встроенных библиотек Python, их читать не нужно, ищите глазами пути к своим файлам вроде `main.py`). Перейдите на указанную строку.
3. **Примените `print()` перед сломанной строкой.** Если упала строка `user["price"]`, добавьте строчкой выше `print(user)` и запустите код еще раз. Вы увидите, какие реальные данные лежат внутри словаря в момент падения, и сразу поймете, почему ключа `price` там не оказалось.

Веб-разработка на Python: Выбор фреймворка

Когда вы переходите к созданию сайтов и веб-сервисов, первая развилка, с которой вы столкнетесь — это выбор между **Django** («всё включено») и **FastAPI** («скорость и гибкость»).

Критерий	Django ☐	FastAPI ⚡
Философия	Monolith («Батарейки в комплекте»). Из коробки есть всё: админка, база данных, авторизация пользователей, защита от хакеров.	Micro-framework («Минимализм»). Дает только ультра-быструю основу для обработки запросов, а всё остальное вы подключаете сами.

Для чего идеален	Крупные консервативные сайты: интернет-магазины, блоги, соцсети, CRM-системы, где нужны стандартные готовые решения.	API-сервисы, микросервисы, мобильные бэкенды, высоконагруженные чаты и интеграции с нейросетями.
Главная фишка	Готовая админ-панель для управления сайтом, которая создается автоматически одной строчкой кода.	Асинхронность из коробки (async/await) и автоматическая генерация документации для вашего кода.

Пример 1: Создаем микро-сайт на FastAPI за 5 шагов

FastAPI невероятно лаконичен. Всё, что мы учили про функции **def**, асинхронность и декораторы, работает здесь на 100%.

Шаг 1. Импортируем инструмент

```
from fastapi import FastAPI
```

Шаг 2. Создаем веб-приложение (объект класса)

```
app = FastAPI()
```

Шаг 3. Используем ДЕКОРАТОР. Он говорит серверу:

"Если пользователь зайдет на главную страницу сайта (путь '/'), запусти функцию ниже".

```
@app.get("/")
```

```
async def home_page():
```

Шаг 4. Асинхронная функция возвращает данные.

FastAPI сам автоматически превратит этот словарь Python в строку формата JSON!

```
return {"status": "success", "message": "Добро пожаловать на мой первый сайт!"}
```

Шаг 5. Как это запустить (в терминале):

```
# uvicorn main:app --reload
```

В браузере по адресу **http://127.0.0.1** пользователь увидит аккуратный **JSON-ответ**. А если зайти на **/docs**, **FastAPI** сам нарисует красивую интерактивную документацию.

Пример 2: Как устроен мир Django

В **Django** код не пишется в одном файле. Когда вы создаете проект командой **django-admin startproject**, фреймворк сам генерирует целую структуру папок и файлов. Логика там строго разделена по архитектуре **MVT (Model-View-Template)**:

1. Файл **models.py** (Модели / База данных):

Здесь вы с помощью **Классов (ООП)** описываете, как выглядит таблица в базе данных. **Django** сам создаст таблицы в SQL, вам не придется писать сырые SQL-запросы!

```
from django.db import models
```

```
class Product(models.Model):
```

```
    name = models.CharField(max_length=100) # Поле для названия  
товара
```

```
    price = models.IntegerField()           # Поле для цены
```

Файл **views.py** (Представления / Логика):

Здесь пишутся стандартные функции **def**, которые принимают запрос от пользователя, достают данные из базы и решают, что показать.

```
from django.shortcuts import render
```

```
from .models import Product
```

```
def catalog(request):
```

```
    all_goods = Product.objects.all() # Достаем ВСЕ товары из базы одной  
командой
```

```
    return render(request, 'catalog.html', {'products': all_goods}) #  
Отправляем данные в HTML-шаблон
```

Файл **catalog.html** (Шаблон / Интерфейс):

Обычная HTML-страница, внутри которой с помощью специальных циклов (очень похожих на цикл **for** в Python) выводятся карточки товаров для покупателя.

Что выбрать?

- Если ваша цель — быстро создать классический сайт (например, доску объявлений или интернет-магазин) с готовой базой данных и панелью администратора за пару дней, ваш выбор — **Django**.

- Если вы хотите строить современные быстрые веб-сервисы, работать с асинхронностью, обслуживать мобильные приложения или делать бэкенд для Telegram-ботов, ваш выбор — **FastAPI**.

