PYTHON
INTERVIEW QUESTIONS

Подборка вопросов по Python для подготовки к техническим собеседованиям

40
вопросов

СОДЕРЖАНИЕ

Вопрос 1

Что такое Python и каковы его основные особенности?

> ***Python*** *— это высокоуровневый интерпретируемый язык программирования общего назначения с акцентом на читаемость кода и простоту синтаксиса.*

Основные особенности:

- **Интерпретируемый**: код выполняется построчно интерпретатором, без необходимости компиляции.
- **Динамическая типизация**: тип переменной определяется автоматически во время выполнения, а не при объявлении.
- **Автоматическое управление памятью**: встроенный сборщик мусора освобождает неиспользуемую память.
- **Кроссплатформенность**: код работает на Windows, macOS и Linux без изменений.
- **Мультипарадигменность**: поддерживает объектно-ориентированный, функциональный и процедурный стили.
- **Обширная стандартная библиотека**: модули для работы с файлами, сетью, JSON и многим другим.

Где применяется:

- **Веб-разработка**: Django, Flask, FastAPI.
- **Анализ данных и ML**: NumPy, Pandas, scikit-learn.
- **Автоматизация и скрипты**: обработка файлов, DevOps, тестирование.

Вопрос 2

Какие основные типы данных существуют в Python?

Python предоставляет несколько встроенных типов данных:

Числовые типы:

- `int` — целые числа произвольной точности: `42`, `-7`, `1_000_000`
- `float` — числа с плавающей точкой: `3.14`, `-0.5`, `1e10`
- `complex` — комплексные числа: `3+4j`

Текстовый тип:

- `str` — строка символов: `"Привет"`, `'Python'`

Логический тип:

- `bool` — принимает значения `True` или `False`

Специальный тип:

- `NoneType` — единственное значение `None`, означающее отсутствие значения

Приведение типов:

```
# Явное преобразование типов
x = int("42")       # str → int
y = float(42)       # int → float
z = str(3.14)       # float → str
w = bool(0)          # int → bool (False)

# Проверка типа
print(type(x))  # <class 'int'>
```

Вопрос 3

В чём разница между списком и кортежем?

Список:

- **Изменяемый**: можно добавлять, удалять и изменять элементы.
- Создаётся с помощью квадратных скобок `[]`.
- Занимает больше памяти из-за возможности изменения размера.

```
fruits = ["яблоко", "банан", "вишня"]
fruits.append("груша")       # Добавление элемента
fruits[0] = "апельсин"       # Изменение элемента
```

Кортеж:

- **Неизменяемый**: после создания нельзя изменить.
- Создаётся с помощью круглых скобок `()`.
- Работает быстрее и занимает меньше памяти.
- Может использоваться как ключ словаря (так как хешируемый).

```
point = (10, 20)
# point[0] = 5  # TypeError — нельзя изменить

# Кортеж как ключ словаря
locations = {(55.75, 37.62): "Москва"}
```

Когда что использовать:

- `list` — когда коллекция будет изменяться (добавление, удаление элементов).
- `tuple` — когда данные должны быть неизменяемыми (координаты, конфигурации, ключи словарей).

Вопрос 4

Чем отличаются set и frozenset?

set (множество):

- **Изменяемая** неупорядоченная коллекция уникальных элементов.
- Поддерживает добавление и удаление элементов.
- Не может быть ключом словаря или элементом другого множества.

```
colors = {"красный", "зелёный", "синий"}
colors.add("жёлтый")
colors.discard("красный")
```

frozenset (замороженное множество):

- **Неизменяемая** версия множества.
- Не поддерживает добавление и удаление элементов.
- Является хешируемым — может быть ключом словаря.

```
immutable_set = frozenset([1, 2, 3])
# immutable_set.add(4)  # AttributeError

# frozenset как ключ словаря
cache = {frozenset([1, 2]): "результат"}
```

Общие операции над множествами:

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

a | b   # Объединение: {1, 2, 3, 4, 5, 6}
a & b   # Пересечение: {3, 4}
a - b   # Разность: {1, 2}
a ^ b   # Симметрическая разность: {1, 2, 5, 6}
```

На практике:

`set` используется для быстрого удаления дубликатов и проверки принадлежности (`in` за O(1)). `frozenset` нужен, когда множество должно быть ключом словаря или элементом другого множества.

Вопрос 5

Что такое словарь и как он работает?

> ***Словарь*** *— это изменяемая коллекция пар «ключ — значение», реализованная на основе хеш-таблицы.*

Основные характеристики:

- Доступ, вставка и удаление элементов выполняются за **O(1)** в среднем случае.
- Ключи должны быть **хешируемыми** (строки, числа, кортежи).
- Начиная с Python 3.7, словари **сохраняют порядок вставки**.

```
user = {
    "name": "Анна",
    "age": 25,
    "city": "Москва"
}
```

Основные методы:

```
user["name"]              # Доступ по ключу (KeyError если нет)
user.get("email", "-")    # Безопасный доступ с значением по умолчанию

user.keys()               # Все ключи
user.values()             # Все значения
user.items()              # Пары (ключ, значение)

user.pop("city")          # Удалить и вернуть значение
user.update({"age": 26})  # Обновить значения
```

Создание словаря:

```
# Литерал
d1 = {"a": 1, "b": 2}

# Из списка кортежей
d2 = dict([("a", 1), ("b", 2)])

# С помощью dict comprehension
d3 = {x: x ** 2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Вопрос 6

Что такое изменяемые и неизменяемые типы данных?

> *В Python все объекты делятся на **изменяемые** и **неизменяемые** в зависимости от того, можно ли менять их содержимое после создания.*

Неизменяемые:

- `int`, `float`, `bool`
- `str`
- `tuple`
- `frozenset`

При «изменении» создаётся новый объект:

```
x = 10
print(id(x))  # Например: 140234866357520
x += 1
print(id(x))  # Другой id — это новый объект
```

Изменяемые:

- `list`
- `dict`
- `set`

Объект изменяется «на месте»:

```
lst = [1, 2, 3]
print(id(lst))  # Например: 140234866400064
lst.append(4)
print(id(lst))  # Тот же id — объект изменился
```

Почему это важно:

- **Ключи словаря** могут быть только неизменяемыми объектами.
- **Передача в функции**: изменяемые объекты могут быть изменены внутри функции, что может привести к неожиданным побочным эффектам.
- **Значения по умолчанию**: не стоит использовать изменяемые объекты как значения по умолчанию в функциях.

```
# Распространённая ошибка
def add_item(item, lst=[]):  # Один список на все вызовы!
    lst.append(item)
    return lst

# Правильный подход
def add_item(item, lst=None):
    if lst is None:
        lst = []
    lst.append(item)
    return lst
```

Вопрос 7

Как работает индексация и срезы в Python?

Индексация:

- Элементы нумеруются начиная с `0`.
- Отрицательные индексы считаются с конца: `-1` — последний элемент.

```
text = "Python"
text[0]    # 'P'
text[-1]   # 'n'
text[-2]   # 'o'
```

Срезы:

Синтаксис: `[start:stop:step]`

- `start` — начальный индекс (включительно), по умолчанию `0`.
- `stop` — конечный индекс (не включительно), по умолчанию длина последовательности.
- `step` — шаг, по умолчанию `1`.

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

nums[2:5]     # [2, 3, 4]
nums[:3]      # [0, 1, 2]
nums[7:]      # [7, 8, 9]
nums[::2]     # [0, 2, 4, 6, 8]  – каждый второй
nums[::-1]    # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]  – реверс
```

Срезы работают со строками и кортежами:

```
text = "Hello, World!"
text[7:12]    # 'World'
text[::-1]    # '!dlroW ,olleH'

coords = (10, 20, 30, 40, 50)
coords[1:4]   # (20, 30, 40)
```

Вопрос 8

Что такое f-строки и чем они отличаются от других способов форматирования?

Python предлагает несколько способов форматирования строк:

f-строки (f-strings) — рекомендуемый способ:

Доступны начиная с Python 3.6. Позволяют вставлять выражения прямо в строку.

```
name = "Анна"
age = 25
print(f"Привет, {name}! Тебе {age} лет.")
print(f"Через 5 лет тебе будет {age + 5}.")
print(f"Имя большими буквами: {name.upper()}")
```

Метод .format():

```
print("Привет, {}! Тебе {} лет.".format(name, age))
print("Привет, {0}! {0}, тебе {1} лет.".format(name, age))
```

Оператор % (устаревший):

```
print("Привет, %s! Тебе %d лет." % (name, age))
```

Сравнение:

- **f-строки** — самый читаемый и быстрый способ. Поддерживают любые выражения.
- **.format()** — полезен, когда шаблон строки задаётся заранее.
- **%** — устаревший подход, встречается в старом коде.

Форматирование чисел:

```
pi = 3.14159265
print(f"Пи: {pi:.2f}")          # Пи: 3.14
print(f"Число: {1000000:,}")     # Число: 1,000,000
print(f"Процент: {0.856:.1%}")   # Процент: 85.6%
```

Вопрос 9

Как работают условные конструкции if / elif / else?

Условные конструкции позволяют выполнять разные блоки кода в зависимости от условия.

Синтаксис:

```
age = 18

if age < 13:
    print("Ребёнок")
elif age < 18:
    print("Подросток")
else:
    print("Взрослый")
```

Тернарный оператор:

Краткая запись условия в одну строку:

```
status = "совершеннолетний" if age >= 18 else "несовершеннолетний"
```

Истинные и ложные значения:

В Python следующие значения считаются **ложными**:

- `False`, `None`
- `0`, `0.0`
- Пустые коллекции: `""`, `[]`, `()`, `{}`, `set()`

Всё остальное считается **истинным**:

```
items = []

if items:
    print("Список не пуст")
else:
    print("Список пуст")  # Этот вариант
```

Цепочки сравнений:

Python поддерживает цепочки сравнений:

```
x = 5
if 1 < x < 10:
    print("x в диапазоне от 1 до 10")
```

Вопрос 10

Какие циклы существуют в Python и чем они отличаются?

Цикл for:

Перебирает элементы **итерируемого объекта** (список, строка, range и др.):

```
fruits = ["яблоко", "банан", "вишня"]
for fruit in fruits:
    print(fruit)

# Цикл по диапазону чисел
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4
```

Цикл while:

Выполняется, **пока условие истинно**:

```
count = 0
while count < 3:
    print(count)
    count += 1
```

Управляющие операторы:

- `break` — прерывает цикл.
- `continue` — переходит к следующей итерации.

```
for i in range(10):
    if i == 3:
        continue  # Пропускает 3
    if i == 7:
        break     # Останавливает цикл на 7
    print(i)      # 0, 1, 2, 4, 5, 6
```

Блок else у цикла:

Выполняется, если цикл завершился **без** `break` :

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        # Выполнится, если break не сработал
        print(f"{n} - простое число")
```

Вопрос 11

Что такое функция в Python и как её определить?

> ***Функция*** *— это именованный блок кода, который можно вызывать многократно. Функции помогают избежать дублирования и делают код более читаемым.*

Определение и вызов:

```
def greet(name):
    return f"Привет, {name}!"

message = greet("Анна")
print(message)  # Привет, Анна!
```

Параметры и аргументы:

```
# Значения по умолчанию
def power(base, exponent=2):
    return base ** exponent

power(3)     # 9  (exponent = 2)
power(3, 3)  # 27 (exponent = 3)
```

Именованные аргументы:

```
def create_user(name, age, city="Москва"):
    return {"name": name, "age": age, "city": city}

# Именованные аргументы можно передавать в любом порядке
user = create_user(age=25, name="Анна")
```

Возврат нескольких значений:

```
def min_max(numbers):
    return min(numbers), max(numbers)

lo, hi = min_max([3, 1, 7, 2, 9])
print(lo, hi)  # 1 9
```

Функция без return:

Если `return` отсутствует, функция возвращает `None`:

```
def say_hello(name):
    print(f"Привет, {name}!")

result = say_hello("Мир")
print(result)  # None
```

Вопрос 12

Что такое *args и **kwargs?

*args — произвольное число позиционных аргументов:

*Собирает все лишние позиционные аргументы в **кортеж**:*

```
def total(*args):
    return sum(args)

total(1, 2, 3)      # 6
total(10, 20)        # 30
```

kwargs — произвольное число именованных аргументов:

***Собирает все лишние именованные аргументы в** словарь:*

```
def build_profile(**kwargs):
    return kwargs

build_profile(name="Анна", age=25, city="Москва")
# {'name': 'Анна', 'age': 25, 'city': 'Москва'}
```

Комбинирование:

Порядок параметров в определении функции строго фиксирован: обычные → `*args` → именованные → `**kwargs`:

```
def func(a, b, *args, **kwargs):
    print(f"a={a}, b={b}")
    print(f"args={args}")
    print(f"kwargs={kwargs}")

func(1, 2, 3, 4, x=10, y=20)
# a=1, b=2
# args=(3, 4)
# kwargs={'x': 10, 'y': 20}
```

Распаковка при вызове:

```
def greet(name, age):
    print(f"{name}, {age} лет")

args_list = ["Анна", 25]
greet(*args_list)      # Распаковка списка

kwargs_dict = {"name": "Иван", "age": 30}
greet(**kwargs_dict)   # Распаковка словаря
```

Вопрос 13

Что такое анонимная функция?

> ***Lambda*** *— это анонимная (безымянная) функция, определяемая в одну строку. Она может принимать любое количество аргументов, но содержит только одно выражение.*

Синтаксис:

```
# Обычная функция
def square(x):
    return x ** 2

# Эквивалентная lambda
square = lambda x: x ** 2

square(5)  # 25
```

Использование с sorted:

```
users = [
    {"name": "Анна", "age": 25},
    {"name": "Борис", "age": 30},
    {"name": "Вера", "age": 20},
]

# Сортировка по возрасту
sorted_users = sorted(users, key=lambda u: u["age"])
```

Использование с map и filter:

```
numbers = [1, 2, 3, 4, 5]

squares = list(map(lambda x: x ** 2, numbers))
# [1, 4, 9, 16, 25]

evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4]
```

Ограничения:

- Только **одно выражение** — нельзя использовать многострочную логику, циклы или присваивания.
- Снижает **читаемость** при сложных выражениях — лучше использовать обычную функцию.
- Нет **имени** — сложнее отлаживать (в traceback отображается как `<lambda>`).

Вопрос 14

Что такое области видимости переменных в Python?

> *В Python область видимости определяет, где переменная доступна. Python использует правило **LEGB** для поиска переменных.*

Правило LEGB:

- **L — Local**: переменные внутри текущей функции.
- **E — Enclosing**: переменные во внешней (вложенной) функции.
- **G — Global**: переменные на уровне модуля.
- **B — Built-in**: встроенные имена Python (`print` , `len` , `range`).

```
x = "глобальная"  # Global

def outer():
    x = "внешняя"  # Enclosing

    def inner():
        x = "локальная"  # Local
        print(x)  # "локальная"

    inner()

outer()
```

Ключевое слово global:

Позволяет изменять глобальную переменную внутри функции:

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
print(counter)  # 1
```

Ключевое слово nonlocal:

Позволяет изменять переменную из внешней функции:

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        return count

    return inner

counter = outer()
print(counter())  # 1
print(counter())  # 2
```

Вопрос 15

Что такое генераторы списков и как их использовать?

> ***Генератор списков*** *(list comprehension) — это краткий способ создания нового списка на основе существующей коллекции в одну строку.*

Синтаксис:

```
[выражение for элемент in итерируемый_объект]
```

Примеры:

```
# Квадраты чисел
squares = [x ** 2 for x in range(6)]
# [0, 1, 4, 9, 16, 25]

# Эквивалент с циклом
squares = []
for x in range(6):
    squares.append(x ** 2)
```

С условием (фильтрация):

```
# Только чётные числа
evens = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]
```

С условием if/else (преобразование):

```
labels = ["чёт" if x % 2 == 0 else "нечёт" for x in range(5)]
# ['чёт', 'нечёт', 'чёт', 'нечёт', 'чёт']
```

Вложенные генераторы списков:

```
# Таблица умножения
matrix = [[i * j for j in range(1, 4)] for i in range(1, 4)]
# [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

Аналоги для других типов:

```
# Dict comprehension
squares_dict = {x: x ** 2 for x in range(5)}

# Set comprehension
unique_lengths = {len(word) for word in ["кот", "собака", "лис"]}
```

Важно:

Не стоит злоупотреблять сложными вложенными генераторами списков — если выражение плохо читается, лучше использовать обычный цикл.

Вопрос 16

Что такое генераторы и чем они отличаются от списков?

> *Генератор — это функция, которая возвращает элементы по одному с помощью `yield`, а не создаёт весь список в памяти сразу.*

Функция-генератор:

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for num in count_up_to(5):
    print(num)  # 1, 2, 3, 4, 5
```

Генераторное выражение:

Аналог list comprehension, но с круглыми скобками:

```
# List comprehension — создаёт весь список в памяти
squares_list = [x ** 2 for x in range(1000000)]

# Генераторное выражение — вычисляет по одному
squares_gen = (x ** 2 for x in range(1000000))
```

Ключевые отличия от списка:

- **Память**: генератор хранит только текущий элемент, а не всю коллекцию.
- **Одноразовость**: генератор можно пройти только один раз.
- **Ленивость**: элементы вычисляются по запросу, а не заранее.

```
gen = (x for x in range(3))
print(list(gen))  # [0, 1, 2]
print(list(gen))  # [] — уже исчерпан
```

Когда использовать:

- **Генератор** — при работе с большими данными, потоками, когда не нужно хранить все элементы.
- **Список** — когда нужен многократный доступ, индексация или известно, что данных немного.

Вопрос 17

Как работают встроенные функции map, filter и zip?

map — применяет функцию к каждому элементу:

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))
# [1, 4, 9, 16, 25]

# Эквивалент через comprehension
squares = [x ** 2 for x in numbers]
```

filter — отбирает элементы по условию:

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4, 6]

# Эквивалент через comprehension
evens = [x for x in numbers if x % 2 == 0]
```

zip — объединяет несколько коллекций:

```
names = ["Анна", "Борис", "Вера"]
ages = [25, 30, 22]

pairs = list(zip(names, ages))
# [('Анна', 25), ('Борис', 30), ('Вера', 22)]

# Часто используется для создания словаря
user_ages = dict(zip(names, ages))
# {'Анна': 25, 'Борис': 30, 'Вера': 22}
```

Особенности:

- Все три функции возвращают **итераторы**, а не списки — для получения списка нужно обернуть в `list()`.
- `zip` останавливается по **самой короткой** коллекции.

- В большинстве случаев **list comprehension** считается более читаемой альтернативой `map` и `filter`.

Вопрос 18

Что такое распаковка в Python?

> ***Распаковка*** *— это механизм, позволяющий «разобрать» коллекцию на отдельные переменные.*

Множественное присваивание:

```
a, b, c = [1, 2, 3]
print(a, b, c)  # 1 2 3

# Работает с кортежами, строками и другими итерируемыми
x, y = (10, 20)
first, second, third = "abc"
```

Обмен переменных:

```
a, b = 1, 2
a, b = b, a
print(a, b)  # 2 1
```

Распаковка с * (звёздочкой):

Собирает «остаток» элементов в список:

```
first, *rest = [1, 2, 3, 4, 5]
print(first)  # 1
print(rest)   # [2, 3, 4, 5]

first, *middle, last = [1, 2, 3, 4, 5]
print(middle)  # [2, 3, 4]
```

Распаковка в вызовах функций:

```
def greet(name, age, city):
    print(f"{name}, {age}, {city}")

data = ["Анна", 25, "Москва"]
greet(*data)  # Распаковка списка

info = {"name": "Иван", "age": 30, "city": "Питер"}
greet(**info)  # Распаковка словаря
```

Распаковка во вложенных структурах:

```
points = [(1, 2), (3, 4), (5, 6)]
for x, y in points:
    print(f"x={x}, y={y}")
```

Вопрос 19

Как работает обработка исключений в Python?

> ***Исключение*** *— это ошибка, возникающая во время выполнения программы. Python позволяет перехватывать и обрабатывать исключения, чтобы программа не завершалась аварийно.*

Синтаксис try/except:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Деление на ноль!")
```

Полная конструкция try/except/else/finally:

```
try:
    number = int(input("Введите число: "))
except ValueError:
    print("Это не число!")
else:
    # Выполняется, если исключение НЕ произошло
    print(f"Вы ввели: {number}")
finally:
    # Выполняется ВСЕГДА
    print("Завершение работы")
```

Перехват нескольких исключений:

```
try:
    value = int("abc")
except (ValueError, TypeError) as e:
    print(f"Ошибка: {e}")
```

Иерархия основных исключений:

- `BaseException`

- `Exception` — базовый класс для большинства исключений - `ValueError` — неверное значение - `TypeError` — неверный тип - `KeyError` — ключ не найден в словаре - `IndexError`

— индекс за пределами списка - `FileNotFoundError` — файл не найден - `ZeroDivisionError` — деление на ноль

Важное правило:

Перехватывайте **конкретные** исключения вместо общего `except Exception` — это помогает не скрывать неожиданные ошибки.

Вопрос 20

Как создать собственное исключение?

> *Собственные исключения создаются путём наследования от класса `Exception` (или его подклассов).*

Простое пользовательское исключение:

```
class InsufficientFundsError(Exception):
    pass

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError("Недостаточно средств")
    return balance - amount

try:
    withdraw(100, 200)
except InsufficientFundsError as e:
    print(e)  # Недостаточно средств
```

Исключение с дополнительными данными:

```
class ValidationError(Exception):
    def __init__(self, field, message):
        self.field = field
        self.message = message
        super().__init__(f"{field}: {message}")

try:
    raise ValidationError("email", "Некорректный формат")
except ValidationError as e:
    print(e.field)    # email
    print(e.message)  # Некорректный формат
```

Когда создавать свои исключения:

- Когда встроенные исключения не описывают ошибку достаточно точно.
- Для разделения бизнес-логики ошибок (например, `UserNotFoundError`, `PermissionDeniedError`).
- Для удобного перехвата группы связанных ошибок через общий базовый класс.

```
class AppError(Exception):
    """Базовое исключение приложения"""
    pass

class NotFoundError(AppError):
    pass

class AccessDeniedError(AppError):
    pass
```

Вопрос 21

Что такое контекстный менеджер и как работает оператор with?

> ***Контекстный менеджер*** *— это объект, который автоматически выполняет действия при входе в блок кода и при выходе из него (даже при возникновении ошибки).*

Оператор with:

Самый частый пример — работа с файлами:

```
# Без with — нужно не забыть закрыть файл
file = open("data.txt", "r")
try:
    content = file.read()
finally:
    file.close()

# С with — файл закрывается автоматически
with open("data.txt", "r") as file:
    content = file.read()
# Файл уже закрыт
```

Как работает:

- `__enter__()` — вызывается при входе в блок `with`. Возвращаемое значение присваивается переменной после `as`.
- `__exit__()` — вызывается при выходе из блока `with`, **даже если произошло исключение**.

Создание своего контекстного менеджера:

```
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        import time
        elapsed = time.time() - self.start
        print(f"Время выполнения: {elapsed:.2f} сек")
        return False  # Не подавлять исключения

with Timer():
    total = sum(range(1_000_000))
# Время выполнения: 0.03 сек
```

Где используются контекстные менеджеры:

- Работа с **файлами** (`open`)
- **Блокировки** в многопоточности (`threading.Lock`)
- **Подключения к БД** (автоматический commit/rollback)
- **Временные ресурсы** (временные файлы, сетевые соединения)

Вопрос 22

Как читать и записывать файлы в Python?

Открытие файла:

Функция `open()` принимает путь к файлу и режим:

- `'r'` — чтение (по умолчанию)
- `'w'` — запись (перезаписывает файл)
- `'a'` — дозапись в конец файла
- `'rb'` / `'wb'` — чтение / запись в бинарном режиме

Чтение файла:

```
# Прочитать весь файл
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()

# Прочитать построчно
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())

# Прочитать все строки в список
with open("data.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()
```

Запись в файл:

```
# Перезаписать файл
with open("output.txt", "w", encoding="utf-8") as f:
    f.write("Первая строка\n")
    f.write("Вторая строка\n")

# Дозаписать в конец
with open("output.txt", "a", encoding="utf-8") as f:
    f.write("Ещё одна строка\n")
```

Кодировки:

Всегда указывайте `encoding="utf-8"`, чтобы избежать проблем с кириллицей и спецсимволами:

```
# Без указания кодировки может возникнуть UnicodeDecodeError
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()
```

Важно:

Всегда используйте конструкцию `with` — она гарантирует закрытие файла даже при возникновении ошибки.

Вопрос 23

Что такое классы и объекты в Python?

> *Класс — это шаблон (чертёж) для создания объектов.* ***Объект*** *— это конкретный экземпляр класса.*

Определение класса:

```
class Dog:
    # Атрибут класса (общий для всех экземпляров)
    species = "Canis familiaris"

    # Конструктор — вызывается при создании объекта
    def __init__(self, name, age):
        # Атрибуты экземпляра (уникальны для каждого объекта)
        self.name = name
        self.age = age

    # Метод экземпляра
    def bark(self):
        return f"{self.name} говорит: Гав!"
```

Создание объектов:

```
dog1 = Dog("Бобик", 3)
dog2 = Dog("Шарик", 5)

print(dog1.name)      # Бобик
print(dog2.bark())    # Шарик говорит: Гав!
print(dog1.species)   # Canis familiaris
```

self:

- `self` — ссылка на текущий экземпляр класса.
- Передаётся автоматически при вызове метода.
- Через `self` доступны атрибуты и методы объекта.

Атрибуты класса vs экземпляра:

```
class Counter:
    count = 0  # Атрибут класса – общий для всех

    def __init__(self):
        Counter.count += 1  # Изменяем атрибут класса
        self.id = Counter.count  # Атрибут экземпляра

c1 = Counter()
c2 = Counter()
print(Counter.count)  # 2
print(c1.id, c2.id)   # 1 2
```

Вопрос 24

Что такое наследование и как оно работает в Python?

> ***Наследование*** *— это механизм ООП, при котором дочерний класс получает атрибуты и методы родительского класса.*

Базовое наследование:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} издаёт звук"

class Dog(Animal):
    def speak(self):
        return f"{self.name} говорит: Гав!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} говорит: Мяу!"

dog = Dog("Бобик")
print(dog.speak())  # Бобик говорит: Гав!
```

super() — вызов родительского метода:

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)  # Вызов __init__ родителя
        self.breed = breed

dog = Dog("Бобик", 3, "Лабрадор")
print(dog.name, dog.breed)  # Бобик Лабрадор
```

Проверка наследования:

```
print(isinstance(dog, Dog))     # True
print(isinstance(dog, Animal))  # True
print(issubclass(Dog, Animal))  # True
```

Переопределение методов:

Дочерний класс может заменить или дополнить метод родителя:

```
class Shape:
    def area(self):
        return 0

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rect = Rectangle(5, 3)
print(rect.area())  # 15
```

Множественное наследование:

Python поддерживает наследование от нескольких классов. Порядок поиска методов определяется алгоритмом MRO (Method Resolution Order), посмотреть его можно через `ClassName.mro()`.

Вопрос 25

Что такое инкапсуляция в Python?

> ***Инкапсуляция*** *— это принцип ООП, при котором внутренние данные объекта скрываются от прямого доступа извне.*

Конвенции именования:

В Python нет строгих модификаторов доступа (`private` , `public`). Вместо этого используются конвенции:

- `name` — публичный атрибут, доступен всем.
- `_name` — «защищённый» (по конвенции), предназначен для внутреннего использования.
- `__name` — «приватный», Python применяет **name mangling** (изменение имени).

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # «Приватный» атрибут

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
# print(account.__balance)  # AttributeError
print(account.get_balance())  # 1000

# Name mangling – атрибут доступен через изменённое имя
print(account._BankAccount__balance)  # 1000
```

@property — управляемый доступ:

Позволяет использовать методы как атрибуты:

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Ниже абсолютного нуля!")
        self._celsius = value

    @property
    def fahrenheit(self):
        return self._celsius * 9 / 5 + 32

temp = Temperature(25)
print(temp.celsius)     # 25
print(temp.fahrenheit)  # 77.0
temp.celsius = 30       # Использует setter
```

Вопрос 26

Что такое полиморфизм в Python?

> ***Полиморфизм*** *— это способность объектов разных классов реагировать на одинаковый вызов по-разному.*

Полиморфизм через переопределение методов:

```
class Cat:
    def speak(self):
        return "Мяу!"

class Dog:
    def speak(self):
        return "Гав!"

class Duck:
    def speak(self):
        return "Кря!"

# Одинаковый интерфейс, разное поведение
animals = [Cat(), Dog(), Duck()]
for animal in animals:
    print(animal.speak())
```

Утиная типизация:

> *«Если это выглядит как утка, плавает как утка и крякает как утка — значит, это утка.»*

Python не проверяет тип объекта — важно лишь наличие нужного метода:

```
class File:
    def read(self):
        return "данные из файла"

class Database:
    def read(self):
        return "данные из БД"

def load_data(source):
    # Не важно, какой тип — важно, что есть метод read
    return source.read()

print(load_data(File()))      # данные из файла
print(load_data(Database()))  # данные из БД
```

Полиморфизм встроенных функций:

```
# len() работает с разными типами
print(len("Python"))    # 6
print(len([1, 2, 3]))   # 3
print(len({"a": 1}))    # 1

# + работает по-разному
print(1 + 2)           # 3 (сложение)
print("Привет, " + "мир!")  # Привет, мир! (конкатенация)
```

Вопрос 27

Что такое статические методы и методы класса?

Обычный метод (метод экземпляра):

Получает ссылку на экземпляр (`self`) в качестве первого аргумента:

```
class MyClass:
    def instance_method(self):
        return f"Вызван для {self}"
```

@classmethod — метод класса:

Получает ссылку на **класс** (`cls`) вместо экземпляра:

```
class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_string(cls, data_string):
        name, age = data_string.split(",")
        return cls(name, int(age))

# Альтернативный конструктор
user = User.from_string("Анна,25")
print(user.name)  # Анна
```

@staticmethod — статический метод:

Не получает ни `self` , ни `cls` . По сути — обычная функция внутри класса:

```
class MathUtils:
    @staticmethod
    def is_even(n):
        return n % 2 == 0

print(MathUtils.is_even(4))  # True
```

Когда что использовать:

- **Метод экземпляра** — когда нужен доступ к атрибутам объекта (`self`).

- **@classmethod** — для альтернативных конструкторов или работы с атрибутами класса.
- **@staticmethod** — для утилитных функций, логически связанных с классом, но не зависящих от экземпляра или класса.

Вопрос 28

Что такое магические методы?

> ***Магические методы*** *(от англ. double underscore — двойное подчёркивание) — это специальные методы, которые определяют поведение объектов при стандартных операциях.*

Строковое представление:

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __str__(self):
        # Для пользователя (print, str())
        return f"{self.name}: {self.price} руб."

    def __repr__(self):
        # Для разработчика (отладка, repr())
        return f"Product('{self.name}', {self.price})"

p = Product("Книга", 500)
print(p)       # Книга: 500 руб.
print(repr(p)) # Product('Книга', 500)
```

Сравнение объектов:

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

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        return (self.x ** 2 + self.y ** 2) < (other.x ** 2 + other.y ** 2)

Point(1, 2) == Point(1, 2)  # True
Point(1, 2) < Point(3, 4)   # True
```

Другие полезные магические методы:

- `__len__` — поведение `len(obj)`
- `__getitem__` — доступ по индексу `obj[key]`
- `__contains__` — оператор `in`
- `__add__` — оператор `+`
- `__call__` — вызов объекта как функции `obj()`

```
class Basket:
    def __init__(self):
        self.items = []

    def __len__(self):
        return len(self.items)

    def __contains__(self, item):
        return item in self.items

basket = Basket()
basket.items.append("яблоко")
print(len(basket))           # 1
print("яблоко" in basket)    # True
```

Вопрос 29

Что такое декораторы и как они работают?

> ***Декоратор*** *— это функция, которая принимает другую функцию и возвращает новую, расширяя её поведение без изменения исходного кода.*

Простой декоратор:

```
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"Вызов функции: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Результат: {result}")
        return result
    return wrapper

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

add(3, 5)
# Вызов функции: add
# Результат: 8
```

Как это работает:

```
# Запись с @decorator
@log_call
def add(a, b):
    return a + b

# Эквивалентна
def add(a, b):
    return a + b
add = log_call(add)
```

Сохранение метаданных (@wraps):

```
from functools import wraps

def log_call(func):
    @wraps(func)  # Сохраняет имя и docstring оригинала
    def wrapper(*args, **kwargs):
        print(f"Вызов: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_call
def add(a, b):
    """Складывает два числа."""
    return a + b

print(add.__name__)  # add (без @wraps было бы wrapper)
print(add.__doc__)   # Складывает два числа.
```

Практические примеры декораторов:

- Логирование вызовов функций
- Замер времени выполнения
- Кеширование результатов
- Проверка прав доступа
- Валидация аргументов

Вопрос 30

Что такое абстрактные классы и зачем они нужны?

> ***Абстрактный класс*** *— это класс, который нельзя создать напрямую. Он задаёт интерфейс (набор обязательных методов) для дочерних классов.*

Модуль abc:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """Вычислить площадь фигуры"""
        pass

    @abstractmethod
    def perimeter(self):
        """Вычислить периметр фигуры"""
        pass

# shape = Shape()  # TypeError: нельзя создать экземпляр
```

Реализация абстрактного класса:

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        import math
        return math.pi * self.radius ** 2

    def perimeter(self):
        import math
        return 2 * math.pi * self.radius

rect = Rectangle(5, 3)
print(rect.area())       # 15
print(rect.perimeter())  # 16
```

Зачем нужны:

- **Контракт**: гарантируют, что дочерний класс реализует все необходимые методы.
- **Документация**: явно показывают, какие методы должны быть реализованы.
- **Раннее обнаружение ошибок**: если метод не реализован, Python выдаст `TypeError` при создании экземпляра, а не при вызове метода.

Вопрос 31

Как работает импорт модулей в Python?

> ***Модуль*** *— это файл с расширением* `.py` *, содержащий Python-код (функции, классы, переменные). Модули позволяют организовать код и переиспользовать его.*

Способы импорта:

```
# Импорт всего модуля
import math
print(math.sqrt(16))  # 4.0

# Импорт конкретных объектов
from math import sqrt, pi
print(sqrt(16))  # 4.0

# Импорт с псевдонимом
import datetime as dt
now = dt.datetime.now()
```

Свой модуль:

```
# utils.py
def greet(name):
    return f"Привет, {name}!"

# main.py
from utils import greet
print(greet("Мир"))
```

__name__ == "__main__":

Позволяет отличить, запущен ли файл напрямую или импортирован как модуль:

```
# my_module.py
def main():
    print("Запуск модуля напрямую")

if __name__ == "__main__":
    # Этот код выполнится только при прямом запуске:
    # python my_module.py
    main()
```

Пакеты:

Пакет — это папка с файлом `__init__.py`, содержащая несколько модулей:

```
my_package/
    __init__.py
    module_a.py
    module_b.py
```

```
from my_package.module_a import some_function
```

Вопрос 32

Что такое виртуальное окружение и зачем оно нужно?

> ***Виртуальное окружение*** *— это изолированная среда Python, в которой устанавливаются пакеты только для конкретного проекта, не влияя на глобальную установку.*

Зачем нужно:

- Разные проекты могут требовать **разные версии** одной библиотеки.
- Избежать **конфликтов** между зависимостями проектов.
- Зафиксировать точные версии пакетов для **воспроизводимости**.

Создание и использование:

```
# Создание виртуального окружения
python -m venv venv

# Активация
# macOS/Linux:
source venv/bin/activate
# Windows:
venv\Scripts\activate

# Деактивация
deactivate
```

Управление пакетами:

```
# Установка пакета
pip install requests

# Установка конкретной версии
pip install requests==2.31.0

# Просмотр установленных пакетов
pip list

# Сохранение зависимостей в файл
pip freeze > requirements.txt

# Установка зависимостей из файла
pip install -r requirements.txt
```

requirements.txt:

```
requests==2.31.0
flask==3.0.0
pytest==7.4.3
```

Этот файл фиксирует все зависимости проекта и позволяет воспроизвести окружение на другой машине.

Вопрос 33

Как работать с JSON в Python?

> ***JSON** (JavaScript Object Notation) — это текстовый формат обмена данными. Python предоставляет встроенный модуль `json` для работы с ним.*

Основные функции:

- `json.dumps()` — Python-объект → JSON-строка
- `json.loads()` — JSON-строка → Python-объект
- `json.dump()` — Python-объект → JSON-файл
- `json.load()` — JSON-файл → Python-объект

Сериализация (Python → JSON):

```
import json

data = {
    "name": "Анна",
    "age": 25,
    "hobbies": ["чтение", "плавание"],
    "active": True
}

# В строку
json_string = json.dumps(data, ensure_ascii=False, indent=2)
print(json_string)

# В файл
with open("data.json", "w", encoding="utf-8") as f:
    json.dump(data, f, ensure_ascii=False, indent=2)
```

Десериализация (JSON → Python):

```
# Из строки
json_string = '{"name": "Анна", "age": 25}'
data = json.loads(json_string)
print(data["name"])  # Анна

# Из файла
with open("data.json", "r", encoding="utf-8") as f:
    data = json.load(f)
```

Соответствие типов:

JSON	Python
object	dict
array	list
string	str
number	int/float
true/false	True/False
null	None

Вопрос 34

Что такое аннотации типов в Python?

> ***Аннотации типов*** *— это способ указать ожидаемые типы аргументов функций, возвращаемых значений и переменных. Они* ***не влияют*** *на выполнение программы, но помогают в разработке.*

Базовый синтаксис:

```
def greet(name: str) -> str:
    return f"Привет, {name}!"

age: int = 25
price: float = 19.99
is_active: bool = True
```

Коллекции:

```
# Python 3.9+
def process(items: list[str]) -> dict[str, int]:
    return {item: len(item) for item in items}

# Вложенные типы
matrix: list[list[int]] = [[1, 2], [3, 4]]
```

Модуль typing:

```
from typing import Optional, Union

# Может быть str или None
def find_user(user_id: int) -> Optional[str]:
    if user_id == 1:
        return "Анна"
    return None

# Может быть int или str
def parse(value: Union[int, str]) -> str:
    return str(value)
```

Зачем нужны аннотации:

- **Документация**: тип ясен без чтения тела функции.
- **IDE**: автодополнение и подсказки работают лучше.
- **Поиск ошибок**: инструменты вроде `mypy` находят ошибки типов до запуска кода.
- **Командная работа**: упрощают понимание кода другими разработчиками.

Важно:

Аннотации — это **подсказки**, а не ограничения. Python **не проверяет** типы во время выполнения:

```
def add(a: int, b: int) -> int:
    return a + b

add("hello", " world")  # Сработает без ошибки: "hello world"
```

Вопрос 35

Чем отличается глубокое копирование от поверхностного?

Поверхностное копирование:

Создаёт новый объект, но **вложенные объекты не копируются** — сохраняются ссылки на оригиналы:

```
import copy

original = [[1, 2, 3], [4, 5, 6]]
shallow = copy.copy(original)

shallow[0][0] = 999
print(original[0][0])  # 999 — изменился и оригинал!
```

Глубокое копирование:

Создаёт **полностью независимую копию**, включая все вложенные объекты:

```
import copy

original = [[1, 2, 3], [4, 5, 6]]
deep = copy.deepcopy(original)

deep[0][0] = 999
print(original[0][0])  # 1 — оригинал не изменился
```

Способы поверхностного копирования:

```
# Для списков
lst = [1, 2, 3]
copy1 = lst.copy()
copy2 = lst[:]
copy3 = list(lst)

# Для словарей
d = {"a": 1}
copy4 = d.copy()
copy5 = dict(d)

# Универсальный
import copy
copy6 = copy.copy(lst)
```

Когда что использовать:

- **Поверхностное** — когда коллекция содержит только неизменяемые элементы (числа, строки).
- **Глубокое** — когда есть вложенные изменяемые объекты (списки в списках, словари в списках).

```
# Поверхностного достаточно
nums = [1, 2, 3]  # Элементы — неизменяемые int
safe_copy = nums.copy()

# Нужно глубокое
matrix = [[1, 2], [3, 4]]  # Вложенные списки
safe_copy = copy.deepcopy(matrix)
```

Вопрос 36

Что такое итераторы и протокол итерации?

> ***Итератор*** *— это объект, который возвращает элементы по одному через метод* `__next__()` *и сигнализирует об окончании через исключение* `StopIteration`.

Протокол итерации:

- `__iter__()` — возвращает сам итератор.
- `__next__()` — возвращает следующий элемент или выбрасывает `StopIteration`.

Как работает цикл for:

```
# Что делает for под капотом
nums = [1, 2, 3]

# for num in nums:
#     print(num)

# Эквивалент:
iterator = iter(nums)       # Вызывает nums.__iter__()
while True:
    try:
        num = next(iterator)  # Вызывает iterator.__next__()
        print(num)
    except StopIteration:
        break
```

Создание своего итератора:

```
class Countdown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value

for num in Countdown(5):
    print(num)  # 5, 4, 3, 2, 1
```

Встроенные итерируемые объекты:

- `list`, `tuple`, `str`, `dict`, `set` — **итерируемые** (имеют `__iter__`).
- `range()`, `map()`, `filter()`, `zip()` — возвращают **итераторы**.
- **Итератор** можно пройти только **один раз**, а **итерируемый объект** можно проходить многократно.

На практике:

Протокол итерации лежит в основе циклов `for`, генераторов и многих встроенных функций. Собственные итераторы полезны, когда нужно обрабатывать данные порциями — например, читать большой файл построчно вместо загрузки целиком.

Вопрос 37

В чём разница между is и ==?

== (сравнение по значению):

Проверяет, **равны ли значения** двух объектов:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True – значения одинаковые
```

is (сравнение по идентичности):

Проверяет, являются ли две переменные **одним и тем же объектом** в памяти:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a is b)  # False – это разные объекты
print(id(a), id(b))  # Разные адреса в памяти

c = a
print(a is c)  # True – c ссылается на тот же объект
```

Кеширование малых целых чисел:

Python кеширует целые числа от **-5 до 256**, поэтому:

```
x = 100
y = 100
print(x is y)  # True – кешированный объект

x = 1000
y = 1000
print(x is y)  # False – разные объекты
```

Правило использования:

- Используйте `is` только для сравнения с `None`:

```
# Правильно
if value is None:
    print("Нет значения")

# Неправильно
if value == None:
    print("Нет значения")
```

- Для сравнения значений **всегда** используйте `==`.
- `is` проверяет `id()` объектов — это полезно только для синглтонов (`None`, `True`, `False`).

Вопрос 38

Как работает управление памятью в Python?

> *Python автоматически управляет памятью, освобождая разработчика от ручного выделения и освобождения памяти.*

Подсчёт ссылок:

Каждый объект хранит **счётчик ссылок** — количество переменных, ссылающихся на него. Когда счётчик достигает **нуля**, объект удаляется:

```
import sys

a = [1, 2, 3]
print(sys.getrefcount(a))  # 2 (a + аргумент функции)

b = a  # Ещё одна ссылка
print(sys.getrefcount(a))  # 3

del b  # Удаляем ссылку
print(sys.getrefcount(a))  # 2
```

Сборщик мусора:

Подсчёт ссылок не справляется с **циклическими ссылками**:

```
# Циклическая ссылка
a = []
b = []
a.append(b)
b.append(a)
del a, b
# Счётчик ссылок не станет 0, но объекты недоступны
```

Для этого Python запускает **сборщик мусора** (модуль `gc`), который находит и удаляет такие циклы.

Основные моменты:

- **Подсчёт ссылок** — основной механизм, работает мгновенно.
- **Сборщик мусора** — дополнительный механизм для циклических ссылок.
- **del** — удаляет ссылку на объект, а не сам объект.

- Разработчику редко нужно вмешиваться в управление памятью — Python справляется сам.

Вопрос 39

Что такое замыкание в Python?

> ***Замыкание*** *— это вложенная функция, которая «запоминает» переменные из внешней функции, даже после того как внешняя функция завершилась.*

Как это работает:

```
def make_multiplier(factor):
    def multiply(number):
        return number * factor  # factor «захвачен» из внешней функции
    return multiply

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))   # 10
print(triple(5))   # 15
```

Условия для замыкания:

- Есть **вложенная функция**.
- Вложенная функция использует переменные из **внешней функции**.
- Внешняя функция **возвращает** вложенную функцию.

Практические примеры:

```python
# Счётчик
def make_counter(start=0):
    count = start
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

c = make_counter()
print(c())  # 1
print(c())  # 2
print(c())  # 3

# Логгер
def make_logger(prefix):
    def log(message):
        print(f"[{prefix}] {message}")
    return log

error_log = make_logger("ERROR")
error_log("Файл не найден")  # [ERROR] Файл не найден
```

Вопрос 40

Чем отличается многопоточность от многопроцессности в Python?

Многопоточность:

Потоки работают в **одном процессе** и разделяют общую память:

```
import threading

def download(url):
    print(f"Скачиваю {url}")

threads = []
for url in ["url1", "url2", "url3"]:
    t = threading.Thread(target=download, args=(url,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()  # Ожидание завершения всех потоков
```

Многопроцессность:

Каждый процесс имеет **собственную память** и **собственный интерпретатор**:

```
from multiprocessing import Process

def heavy_computation(n):
    return sum(i * i for i in range(n))

processes = []
for n in [10_000_000, 20_000_000]:
    p = Process(target=heavy_computation, args=(n,))
    processes.append(p)
    p.start()

for p in processes:
    p.join()
```

GIL (Global Interpreter Lock):

GIL — это механизм в CPython, который позволяет **только одному потоку** выполнять Python-код одновременно. Из-за этого:

- Потоки **не ускоряют** задачи с интенсивными вычислениями (CPU-bound).
- Потоки **ускоряют** задачи с ожиданием ввода-вывода (I/O-bound): сетевые запросы, чтение файлов.

Когда что использовать:

- **threading** — для I/O-задач: скачивание файлов, API-запросы, работа с БД.
- **multiprocessing** — для CPU-задач: обработка данных, вычисления, алгоритмы.

Критерий	threading	multiprocessing
Память	Общая	Раздельная
GIL	Ограничивает	Не влияет
Накладные расходы	Низкие	Высокие
Лучше для	I/O-bound	CPU-bound