**📌 1. Создание класса Cat**

class Cat:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

* class Cat: — мы создаём класс Cat (кот).
* \_\_init\_\_ — это **конструктор**. Он вызывается при создании нового объекта.
* self.name и self.age — это **атрибуты** объекта (каждый кот будет иметь имя и возраст).
* Когда мы создаём кота, мы должны указать его имя и возраст.

**📌 2. Создание объектов (экземпляров класса)**

cat1 = Cat("Mittens", 3)

cat2 = Cat("Whiskers", 7)

cat3 = Cat("Shadow", 5)

* cat1 → кот по имени **Mittens**, возраст **3**
* cat2 → кот по имени **Whiskers**, возраст **7**
* cat3 → кот по имени **Shadow**, возраст **5**

Теперь у нас есть три объекта класса Cat.

**📌 3. Функция для поиска самого старого кота**

def find\_oldest\_cat(cat\_a, cat\_b, cat\_c):

oldest = cat\_a

if cat\_b.age > oldest.age:

oldest = cat\_b

if cat\_c.age > oldest.age:

oldest = cat\_c

return oldest

* В функцию передаются **три кота**.
* Сначала предполагаем, что старший кот — это cat\_a.
* Потом сравниваем возраст (.age) с другими котами:
  + Если cat\_b старше, то теперь oldest = cat\_b.
  + Если cat\_c старше текущего, то oldest = cat\_c.
* В конце функция возвращает самого старшего кота.

**📌 4. Поиск и вывод самого старого кота**

oldest\_cat = find\_oldest\_cat(cat1, cat2, cat3)

print(f"The oldest cat is {oldest\_cat.name}, and is {oldest\_cat.age} years old.")

* Вызов функции возвращает объект самого старого кота (в данном случае это **Whiskers, 7 лет**).
* oldest\_cat.name → берём имя кота.
* oldest\_cat.age → берём возраст кота.
* f"..." — f-строка, подставляет значения прямо в строку.

👉 Результат на экране:

The oldest cat is Whiskers, and is 7 years old.

⚡ Итог:

* Ты создал класс Cat с атрибутами name и age.
* Создал 3 объекта (3 кота).
* Сделал функцию, которая находит самого старшего кота.
* Напечатал имя и возраст самого старшего кота.

**Step 1: Создание класса Dog**

class Dog:

def \_\_init\_\_(self, name, height):

self.name = name

self.height = height

* Ключевое слово class создаёт **класс** Dog — "шаблон" для объектов собак.
* Метод \_\_init\_\_ — это **конструктор**: он автоматически вызывается при создании новой собаки.
* Аргументы:
  + self — ссылка на текущий объект (обязателен в методах класса).
  + name — имя собаки.
  + height — высота собаки (в см).
* self.name = name и self.height = height сохраняют эти значения внутри объекта.

**🔹 Методы класса**

def bark(self):

print(f"{self.name} goes woof!")

def jump(self):

print(f"{self.name} jumps {self.height \* 2} cm high!")

* bark() — собака лает. Выводит текст с её именем.
* jump() — собака прыгает. Высота прыжка рассчитывается как **2 × рост**.

**🔹 Step 2: Создание объектов**

davids\_dog = Dog("Rex", 60)

sarahs\_dog = Dog("Teacup", 20)

* Создаются два объекта (две собаки).
  + У Дэвида собака **Rex** ростом 60 см.
  + У Сары собака **Teacup** ростом 20 см.

**🔹 Step 3: Вывод информации и вызов методов**

print(f"David's dog is named {davids\_dog.name} and is {davids\_dog.height} cm tall.")

davids\_dog.bark()

davids\_dog.jump()

* У Rex печатается имя и рост.
* Потом вызываются методы:
  + davids\_dog.bark() → "Rex goes woof!"
  + davids\_dog.jump() → "Rex jumps 120 cm high!" (60 × 2 = 120).

Аналогично с Teacup:

* "Sarah's dog is named Teacup and is 20 cm tall."
* "Teacup goes woof!"
* "Teacup jumps 40 cm high!"

**🔹 Step 4: Сравнение собак по росту**

if davids\_dog.height > sarahs\_dog.height:

print(f"{davids\_dog.name} is bigger than {sarahs\_dog.name}.")

elif davids\_dog.height < sarahs\_dog.height:

print(f"{sarahs\_dog.name} is bigger than {davids\_dog.name}.")

else:

print(f"{davids\_dog.name} and {sarahs\_dog.name} are the same size!")

* Сравниваются значения height у двух собак.
* Так как 60 > 20 → программа напечатает:  
  **"Rex is bigger than Teacup."**

## 🔹 Первая часть: class Song

class Song:

def \_\_init\_\_(self, lyrics):

self.lyrics = lyrics

def sing\_me\_a\_song(self):

for line in self.lyrics:

print(line)

### Объяснение:

* class Song: — создаётся класс **Song**, который моделирует песню.
* \_\_init\_\_(self, lyrics) — конструктор.
  + При создании объекта мы передаём список строк (lyrics) — это слова песни.
  + Он сохраняется в свойстве self.lyrics.
* sing\_me\_a\_song(self) — метод, который построчно выводит текст песни на экран, проходя по списку строк в self.lyrics.

### Использование:

stairway = Song([

"There’s a lady who's sure",

"all that glitters is gold",

"and she’s buying a stairway to heaven"

])

stairway.sing\_me\_a\_song()

✅ Результат:

There’s a lady who's sure

all that glitters is gold

and she’s buying a stairway to heaven

То есть программа хранит текст песни и выводит его строчка за строчкой.

## 🔹 Вторая часть: class Zoo

class Zoo:

def \_\_init\_\_(self, zoo\_name):

self.zoo\_name = zoo\_name

self.animals = []

* Создаётся класс **Zoo** (зоопарк).
* При инициализации указываем название зоопарка и создаём пустой список animals.

### Методы:

1. **Добавить животное**

def add\_animal(self, new\_animal):

if new\_animal not in self.animals:

self.animals.append(new\_animal)

* Добавляет животное, если его ещё нет в зоопарке.
* Используется проверка if new\_animal not in self.animals.

1. **Показать животных**

def get\_animals(self):

print(f"Animals in {self.zoo\_name}: {self.animals}")

* Выводит список всех животных в данном зоопарке.

1. **Продать (удалить) животное**

def sell\_animal(self, animal\_sold):

if animal\_sold in self.animals:

self.animals.remove(animal\_sold)

* Удаляет животное из списка, если оно есть.

1. **Сортировать и сгруппировать животных**

def sort\_animals(self):

sorted\_animals = sorted(self.animals)

grouped\_animals = {}

for animal in sorted\_animals:

first\_letter = animal[0].upper()

if first\_letter not in grouped\_animals:

grouped\_animals[first\_letter] = []

grouped\_animals[first\_letter].append(animal)

return grouped\_animals

* Сначала сортирует список животных по алфавиту (sorted).
* Потом группирует их по первой букве.
* Использует словарь, где ключ — буква, а значение — список животных.

1. **Показать группы**

def get\_groups(self):

grouped = self.sort\_animals()

for letter, group in grouped.items():

print(f"{letter}: {group}")

* Берёт сгруппированных животных и красиво выводит по группам.

class Parrot():

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin():

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# общая функция

def flying\_test(bird):

bird.fly()

# объекты

blu = Parrot()

peggy = Penguin()

flying\_test(blu) # Parrot can fly

flying\_test(peggy) # Penguin can't fly

🔎 Что здесь происходит

Есть два разных класса: Parrot и Penguin.

У каждого есть методы fly() и swim().

Но они реализованы по-разному (у попугая fly действительно летает, а у пингвина — нет).

Функция flying\_test(bird) принимает объект и вызывает метод fly().

Важно: она не знает заранее, будет это Parrot или Penguin.

Она просто вызывает bird.fly() → а Python сам подставляет нужный метод в зависимости от типа объекта.

Это пример полиморфизма:

«Один интерфейс (метод fly) — разные реализации (у попугая и у пингвина)».

📌 Вывод программы

rust

Копировать код

Parrot can fly

Penguin can't fly

✅ Ключевые идеи

Оба класса имеют одинаковый набор методов (fly, swim) → это делает их взаимозаменяемыми в контексте функций.

Такой подход называют "duck typing" в Python:

«Если что-то ведёт себя как утка (умеет fly), то можно обращаться к нему как к утке».

Благодаря этому мы можем писать универсальные функции, которые работают с разными объектами, если у них есть нужные методы.

**Анализ по шагам**

1. **Класс A**
   * Определяет метод dothis(), который печатает "doing this in A".
2. **Класс B(A)**
   * Наследует A.
   * Не имеет собственных методов, значит, будет использовать всё из A.
3. **Класс C**
   * Определяет свой метод dothis(), печатающий "doing this in C".
   * Он не связан с A.
4. **Класс D(B, C)**
   * Наследует сразу от B и C (множественное наследование).
   * Сам ничего не определяет, значит методы ищутся в родителях.
5. **Создаём экземпляр D и вызываем dothis().**

**📌 Как Python решает, какой метод вызвать?**

Python использует **MRO (Method Resolution Order)**.  
Порядок поиска в D:

D → B → A → C → object

* Сначала смотрим в D (нет метода).
* Потом в B (нет метода).
* Потом в A (нашёл dothis) ✅
* До C Python уже не дойдёт.

**📌 Результат выполнения**

doing this in A

**✅ Выводы**

* D вызывает метод из **A**, а не из C.
* Это из-за порядка наследования: class D(B, C) → приоритет у B, а через него у A.
* Если бы поменяли порядок на class D(C, B), тогда MRO стал бы:
* D → C → B → A → object

и результат был бы:

doing this in C

**Разбор классов**

1. **Класс Book**
   * Базовый класс с атрибутами: title, author, publication\_date, price.
   * Метод present() выводит название книги.
2. **Класс Fiction(Book)**
   * Наследует Book.
   * В \_\_init\_\_ сначала вызывает super().\_\_init\_\_, чтобы инициализировать базовые атрибуты.
   * Добавляет новые атрибуты: genre = 'Fiction' и is\_awesome.
   * В зависимости от is\_awesome выставляет bored = False/True и печатает сообщение.

**📌 Ход выполнения программы**

foundation = Fiction('Asimov', 'Foundation', '1966', '5£', True)

* Создаётся объект Fiction.
* super().\_\_init\_\_ сохраняет title='Asimov', author='Foundation', publication\_date='1966', price='5£'.  
  ⚠️ Здесь, возможно, перепутаны местами title и author, потому что "Asimov" скорее автор, а "Foundation" — название.
* Поскольку is\_awesome=True, печатается:
* Woow this is an awesome book
* Атрибут bored = False.

Далее:

foundation.present() # Title: Asimov

print(foundation.price) # 5£

print(foundation.bored) # False

Теперь второй объект:

boring\_book = Fiction('boring\_guy', 'boring\_title', 'boring\_date', '9000£', False)

* Поскольку is\_awesome=False, печатается:
* I am very bored
* Атрибут bored = True.

print(boring\_book.bored) # True

**📌 Вывод программы**

Woow this is an awesome book

Title: Asimov

5£

False

I am very bored

True

**✅ Итоговый анализ**

* Здесь показано **наследование с расширением функционала**: Fiction использует атрибуты Book, но добавляет свои (genre, is\_awesome, bored).
* super() гарантирует, что базовый конструктор тоже отработает.
* Логика с is\_awesome управляет поведением объекта и выводит разные сообщения.

class Fiction(Book):

def \_\_init\_\_(self, title, author, publication\_date, price, is\_awesome):

super().\_\_init\_\_(title, author, publication\_date, price)

self.genre = 'Fiction'

self.is\_awesome = is\_awesome

...

🔎 Что делает super().\_\_init\_\_

Метод \_\_init\_\_ в Fiction переопределяет \_\_init\_\_ класса Book.

Если бы мы не вызвали super(), то атрибуты title, author, publication, price, которые задаются в Book, не инициализировались бы.

super().\_\_init\_\_(...) вызывает конструктор родительского класса Book и выполняет его код:

python

Копировать код

class Book():

def \_\_init\_\_(self, title, author, publication\_date, price):

self.title = title

self.author = author

self.publication = publication\_date

self.price = price

Именно поэтому в объекте Fiction после создания мы можем обращаться к:

python

Копировать код

foundation.title

foundation.author

foundation.publication

foundation.price

📌 Если убрать super()

python

Копировать код

class Fiction(Book):

def \_\_init\_\_(self, title, author, publication\_date, price, is\_awesome):

self.genre = 'Fiction'

self.is\_awesome = is\_awesome

Теперь у объекта не будет атрибутов title, author, publication, price, и при вызове:

python

Копировать код

foundation.present()

мы получим:

pgsql

Копировать код

AttributeError: 'Fiction' object has no attribute 'title'

✅ Вывод

В данном коде super() нужен для того, чтобы:

Вызвать конструктор базового класса Book.

Сохранить общие атрибуты книги (title, author, publication\_date, price) в объекте Fiction.

Избежать дублирования кода (иначе пришлось бы копировать все четыре присваивания из Book в Fiction).