Python and SQL: intro / SQL platforms

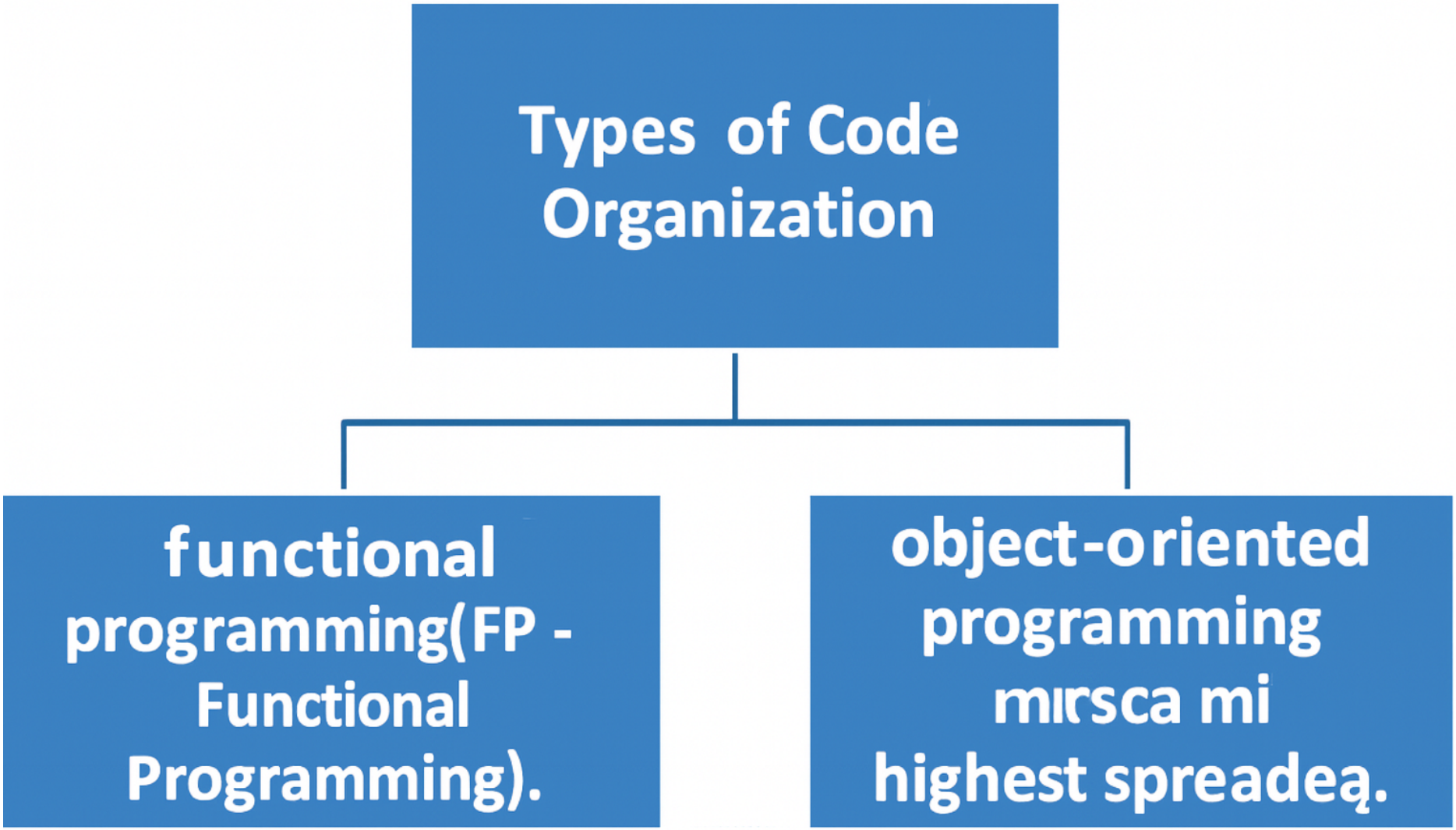
Ewa Weychert

Class 8: Object oriented Programming

UNIWERSYTET WARSZAWSKI | WYDZIAŁ NAUK EKONOMICZNYCH

Class in Python

Classes in Python



OOP vs Functions

```python
def utworz_samochod(marka, model):
    return {"marka": marka, "model": model}

def przedstaw_samochod(samochod):
    return f"Jestem samochodem marki {samochod['marka']}, model {samochod['model']}."

# Tworzenie "obiektu" samochodu jako słownika
auto = utworz_samochod("Toyota", "Corolla")

# Wywołanie funkcji zamiast metody obiektu
print(przedstaw_samochod(auto))
```

```
Jestem samochodem marki Toyota, model Corolla.
```

```python
class Samochod:
    def __init__(self, marka, model):
        self.marka = marka
        self.model = model

    def przedstaw_sie(self):
        return f"Jestem samochodem marki {self.marka}, model {self.model}."

auto = Samochod("Toyota", "Corolla")
print(auto.przedstaw_sie())
```

```
Jestem samochodem marki Toyota, model Corolla.
```

OOP vs FP

Feature	OOP	FP
Data structure	Class `Car`	Dictionary `{brand, model}`
Creating an instance	`car = Car("Toyota", "Corolla")`	`car = create_car("Toyota", "Corolla")`
Accessing data	`car.brand`	`car["brand"]`
Behavior	Methods (`introduce()`)	Functions (`introduce_car(car)`)

OOP is ideal for modeling real-world objects and systems, where their **properties** and **interactions** are important.

FP works great in applications where the key aspect is **data transformation**, e.g. data analysis, mathematical computations.

Comparison: Object-Oriented vs Functional Programming

Feature	**Object-Oriented Programming (OOP)**	**Functional Programming (FP)**
Basic unit	Classes and objects	Functions
Program state	Mutable, stored in objects	Immutable, each function operates on new data
Way of organizing code	Grouping methods and data into classes	Splitting the program into functions performing single operations
Example use cases	Creating desktop applications, data management systems	Processing large datasets, mathematical operations
Approach to the problem	"Who are we?" (e.g. object `Car` has brand, model and methods)	"What do we do?" (e.g. function `double(x)` returns `x * 2`)

Classes in Python

Have you ever come across **classes in Python**? **Yes!** :)

Built-in classes in Python define how an instance of a given class interacts with various methods and objects, defining its **behavior** and **capabilities** within the program.

Each instance of a class **inherits attributes and methods** that allow it to act according to a defined pattern, which makes programming more **modular**, **readable**, and **flexible**.

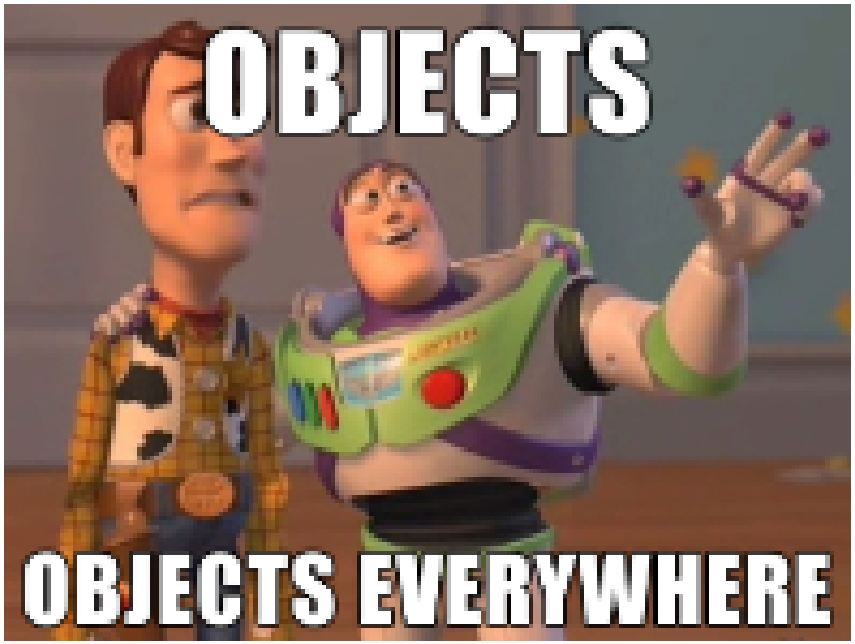
```
y = "hello"
print(type(y))

x = 1
print(type(x))
```

```
<class 'str'>
<class 'int'>
```

```
text = "Python"  # Instancja klasy str
print(type(text))
print(text.upper())  # Zamiana na wielkie litery metoda
```

```
<class 'str'>
PYTHON
```

I see objects everywhere ;)



Class and Object in Python

- A **class** is a template for objects.
- **Objects** are instances belonging to classes.
- An object has structure, which means that it has:
  - **attributes** – properties describing the object,
  - **methods** – behaviors or actions that an object of a given class can perform.
- The behavior of an object consists of the operations it performs.
- Attributes and operations are collectively called the **components of an object**.

```
text = "Python"  # Instancja klasy str
print(type(text))
print(text.upper())  # Zamiana na wielkie litery metoda
```

```
<class 'str'>
PYTHON
```

Class and Object in Python

- `upper()` is a method of the `str` class in Python.
- `upper()` returns the text in **uppercase**.
- Methods are functions assigned to objects that operate on their data.
- Calling `text.upper()` does not modify the original object `text`, but returns a new string.
- The `str` class has no attributes that can be changed – it is **immutable**.

```
text = "Python"  # Instancja klasy str
print(type(text))
print(text.upper())  # Zamiana na wielkie litery metoda
```

```
<class 'str'>
PYTHON
```

Example of a built-in class in Python: `datetime.datetime`

Attributes of the `datetime.datetime` class

Attributes in this class are properties of the object (they store values):

- `year` → year (e.g. 2024)
- `month` → month (e.g. 3)
- `day` → day of the month (e.g. 7)
- `hour` → hour (e.g. 15)
- `minute` → minute (e.g. 30)
- `second` → second (e.g. 45)

Methods of the `datetime.datetime` class Methods allow you to manipulate date and time:

- `now()` → returns the current date and time
- `today()` → returns today's date (without time)
- `weekday()` → returns the day of the week (0 = Monday, 6 = Sunday)
- `strftime(format)` → returns a formatted date/time string

Summary: The `datetime.datetime` class is a built-in class in Python. It has **attributes** (e.g. `year`, `month`, `day`) and **methods** (e.g. `strftime()`, `now()`, `weekday()`).

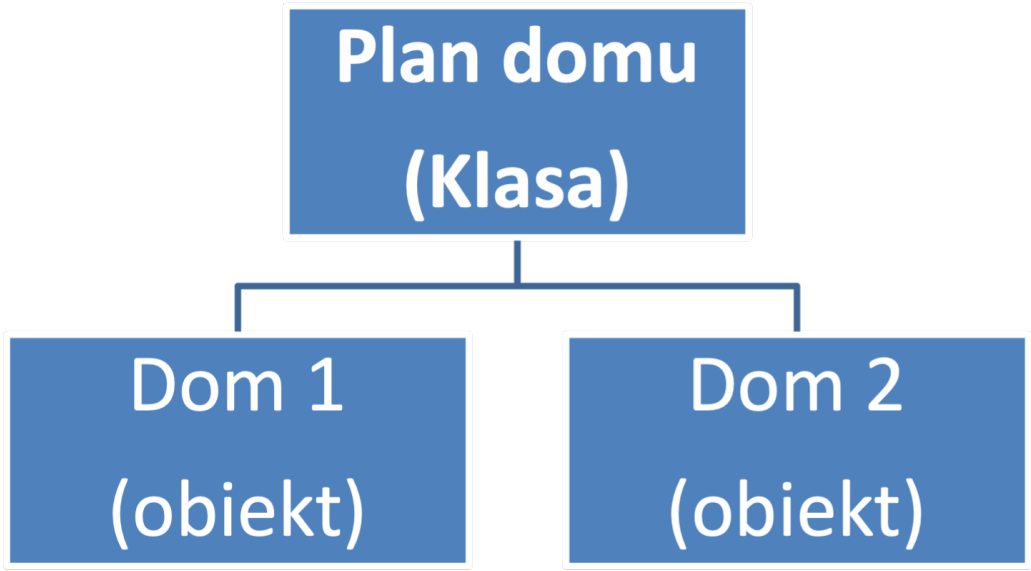
```python
from datetime import datetime

# Tworzymy obiekt daty i czasu
now = datetime.now()

# Atrybuty klasy datetime.datetime
print(now.year)   # Rok
print(now.month)  # Miesiąc
print(now.day)    # Dzień
print(now.hour)   # Godzina
print(now.minute) # Minuta
print(now.second) # Sekunda

# Metody klasy datetime.datetime
print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Formatowanie daty
print(now.weekday())  # Dzień tygodnia (0 = poniedziałek, 6 = niedziela)

print(now.strftime("%d-%m-%Y %H:%M"))  # Format: DD-MM-YYYY HH:MM
```

```
2025
3
7
12
9
26
2025-03-07 12:09:26
4
07-03-2025 12:09
```

Python Classes and Objects

Think of a class as a house blueprint. It contains all details about floors, doors, windows, etc. Based on these descriptions, we build the house.

The actual, physical house is the object. Similarly in Python — the class is the *plan*, and the object is its *instance*.



Example 1 in Python

```
class Student:
    pass
```

Klasa

```
student1 = Student()
student2 = Student()
```

Obiekt

How can we define attributes and methods inside a class?

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

    def check_pass_fail(self):
        if self.marks >= 40:
            return True
        else:
            return False

student1 = Student('Harry', 85)
print(student1.name)
print(student1.marks)
```

Defining attributes and methods in a class:
Attributes can be defined directly when creating an object instance. To do this, we use the `__init__()` method, which initializes a new object of the class.

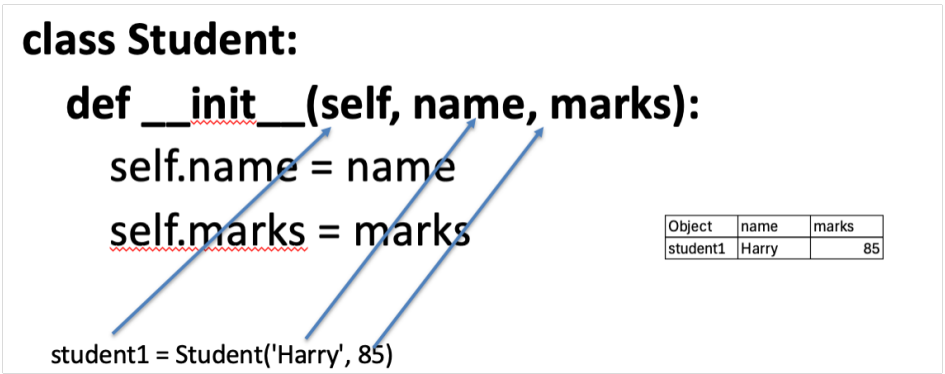
If you use other languages such as **C++** or **Java**, the `__init__()` method in Python is very similar to a **constructor**.

How does the `__init__()` method work in practice

- When we create an object, the `__init__()` method is called **automatically**.
- While creating the object `student1`, we passed the values `'Harry'` and `85` to its `name` and `marks` attributes in the `__init__()` method.
- For the object `student1`:
  - the `name` attribute will be equal to `'Harry'`,
  - the `marks` attribute will be equal to `85`.

Accessing the values:

```
student1.name
student1.marks
```



Creating the `check_pass_fail(self)` method

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

    def check_pass_fail(self):
        if self.marks >= 40:
            return True
        else:
            return False
```

```
# Tworzenie obiektu studenta
student1 = Student("Jan", 45)
student2 = Student("Anna", 35)

# Sprawdzenie, czy zdali
print(student1.name, "Zdał:", student1.check_pass_fail())  # Jan Zdał: True
print(student2.name, "Zdał:", student2.check_pass_fail())  # Anna Zdał: False
```

Methods are functions defined inside a class. They operate on objects of that class and allow performing actions related to their data.

The `check_pass_fail()` method in the `Student` class checks whether a given student has **passed the exam** or not.

Description of the method:

- **Goal:** determine whether the student has achieved a passing score.
- **How it works:** If `marks` $\geq 40 \rightarrow$ the method returns `True` (the student passed), otherwise `False` (the student failed).
- **Use:** You can use this method to easily filter passing and failing students.

Creating a class in Python

A class is created using the `class` keyword. It allows you to define your own data type that combines **attributes** (data) and **methods** (behaviors).

```
class Samochod:
    def __init__(self, marka, model):
        self.marka = marka
        self.model = model
```

Object as an instance of a class

An **object** is an **instance of a class** — a concrete example created based on the class definition. Each object has its own values of the attributes defined in the class.

```
class Samochod:
    def __init__(self, marka, model):
        self.marka = marka
        self.model = model

moj_samochod = Samochod('Toyota', 'Corolla')
print(moj_samochod.marka)
```

Methods in Python classes

Methods are **functions defined inside a class**. They operate on objects of that class and allow performing actions related to their data.

```
class Samochod:
    def __init__(self, marka, model):
        self.marka = marka
        self.model = model

    def pokaz_informacje(self):
        print(f"Marka: {self.marka}, Model: {self.model}")
```

Inheritance

```python
# Definicja klasy Kot
class Kot:
    def __init__(self, imie, wiek):
        """
        Konstruktor klasy Kot
        :param imie: Imię kota
        :param wiek: Wiek kota
        """
        self.imie = imie  # Przypisanie imienia
        self.wiek = wiek  # Przypisanie wieku

    def mow(self):
        """Metoda zwracająca dźwięk kota"""
        print("Miau")


# Definicja klasy Pies
class Pies:
    def __init__(self, imie, wiek):
        """
        Konstruktor klasy Pies
        :param imie: Imię psa
        :param wiek: Wiek psa
        """
        self.imie = imie  # Przypisanie imienia
        self.wiek = wiek  # Przypisanie wieku

    def mow(self):
        """Metoda zwracająca dźwięk psa"""
        print("Hau")


# Przykładowe użycie klas
kot1 = Kot("Mruczek", 3)
pies1 = Pies("Reksio", 5)

# Wywołanie metod dla obiektów
kot1.mow()  # Wypisze: Miau
pies1.mow()  # Wypisze: Hau
```

```
Miau
Hau
```

konstruktor __init__(), który inicjalizuje imie oraz oraz wiek jest taki samw klasie kot i pies

Powtarzamy niepotrzebnie kod

Potencialne rozwiazanie:

dziedziczenie

```python
# Definicja klasy Zwierzak
class Zwierzak:
    def __init__(self, imie, wiek):
        """
        Konstruktor klasy Zwierzak
        :param imie: Imię zwierzaka
        :param wiek: Wiek zwierzaka
        """
        self.imie = imie  # Przypisanie imienia
        self.wiek = wiek  # Przypisanie wieku

    def pokaz(self):
        """Metoda wyświetlająca informacje o zwierzaku"""
        print(f"Jestem {self.imie} i mam {self.wiek} lat(a)")


# Klasa Kot dziedzicząca po klasie Zwierzak
class Kot(Zwierzak):
    def mow(self):
        """Metoda zwracająca dźwięk kota"""
        print("Miau")


# Klasa Pies dziedzicząca po klasie Zwierzak
class Pies(Zwierzak):
    def mow(self):
        """Metoda zwracająca dźwięk psa"""
        print("Hau")
```

Comparison: inheritance vs no inheritance

Feature	**First code (without inheritance)**	**Second code (with inheritance)**
Code organization	Each class defines its own attributes and methods separately.	The `Animal` class stores common attributes.
Avoiding duplication	**NO** – `Cat` and `Dog` repeat the same code (`__init__()`).	**YES** – the `__init__()` constructor is in `Animal`.
Code readability	More repetition, harder to maintain.	More modular and clearer.
Extensibility	You have to manually copy code when adding a new class.	You just create a new class and inherit from `Animal`.
Use case	Good for simple, independent objects.	Better for structures with many common features.

Code comparison — with and without inheritance

Without inheritance

```python
# Definicja klasy Kot
class Kot:
    def __init__(self, imie, wiek):
        """
        Konstruktor klasy Kot
        :param imie: Imię kota
        :param wiek: Wiek kota
        """
        self.imie = imie  # Przypisanie imienia
        self.wiek = wiek  # Przypisanie wieku

    def mow(self):
        """Metoda zwracająca dźwięk kota"""
        print("Miau")


# Definicja klasy Pies
class Pies:
    def __init__(self, imie, wiek):
        """
        Konstruktor klasy Pies
        :param imie: Imię psa
        :param wiek: Wiek psa
        """
        self.imie = imie  # Przypisanie imienia
```

With inheritance

```python
# Definicja klasy Zwierzak
class Zwierzak:
    def __init__(self, imie, wiek):
        """
        Konstruktor klasy Zwierzak
        :param imie: Imię zwierzaka
        :param wiek: Wiek zwierzaka
        """
        self.imie = imie  # Przypisanie imienia
        self.wiek = wiek  # Przypisanie wieku

    def pokaz(self):
        """Metoda wyświetlająca informacje o zwierzaku"""
        print(f"Jestem {self.imie} i mam {self.wiek} lat(a)")


# Klasa Kot dziedzicząca po klasie Zwierzak
class Kot(Zwierzak):
    def mow(self):
        """Metoda zwracająca dźwięk kota"""
        print("Miau")


# Klasa Pies dziedzicząca po klasie Zwierzak
class Pies(Zwierzak):
    def mow(self):
        """Metoda zwracająca dźwięk psa"""
        print("Hau")
```

Comparison: without inheritance vs with inheritance

Without inheritance:

- `Cat` and `Dog` are separate, independent classes.
- Each class has its own `__init__()` constructor that initializes `imie` and `wiek`.
- Both classes have a `mow()` method, but their code is not shared.

With inheritance:

- `Cat` and `Dog` inherit from the common `Animal` class.
- The `Animal` class stores common attributes (`imie`, `wiek`) and the `pokaz()` method.
- `Cat` and `Dog` inherit everything from `Animal`, but add their own `mow()` method.

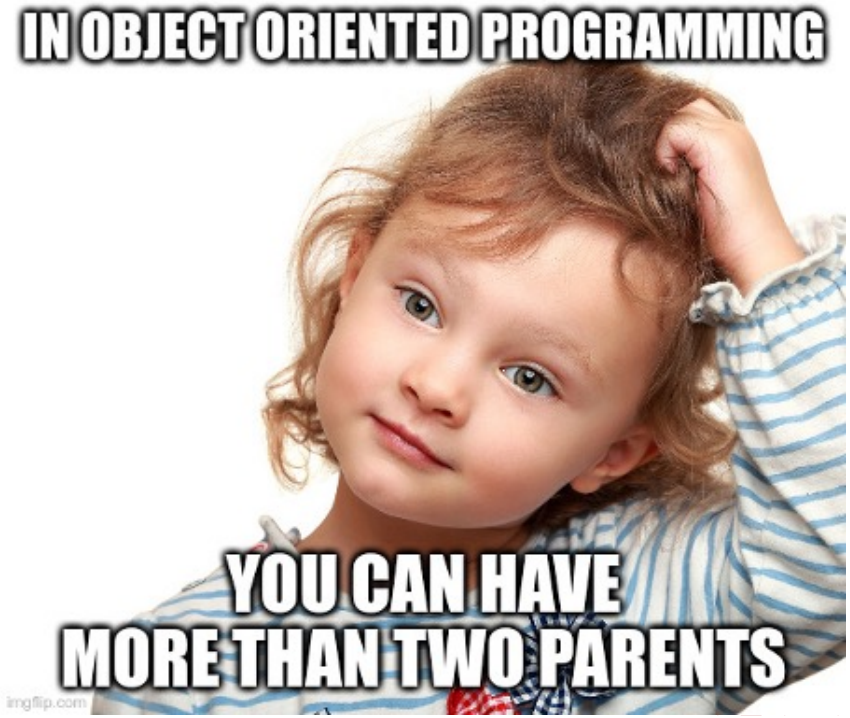
What is inheritance?

Inheritance is a mechanism that allows one class (`Cat`, `Dog`) to take over properties and methods of another class (`Animal`). Thanks to this:

- you avoid code duplication — common features are written only once,
- the code becomes more modular and easier to extend.

Multiple inheritance

- Yes, in object-oriented programming (OOP) in Python you can have more than two parents — this is called **multiple inheritance**.
- It allows a class to **inherit methods and attributes** from more than one base class.
- This way you can combine functionality from different classes, but it requires caution to avoid name conflicts and inheritance errors.



How does multiple inheritance work?

The `Papuga` (Parrot) class inherits from three classes: `Zwierzak` (Animal), `Latajacy` (Flying) and `Ptak` (Bird). Thanks to this, an object of the `Papuga` class can use methods of all its parents.

Example:

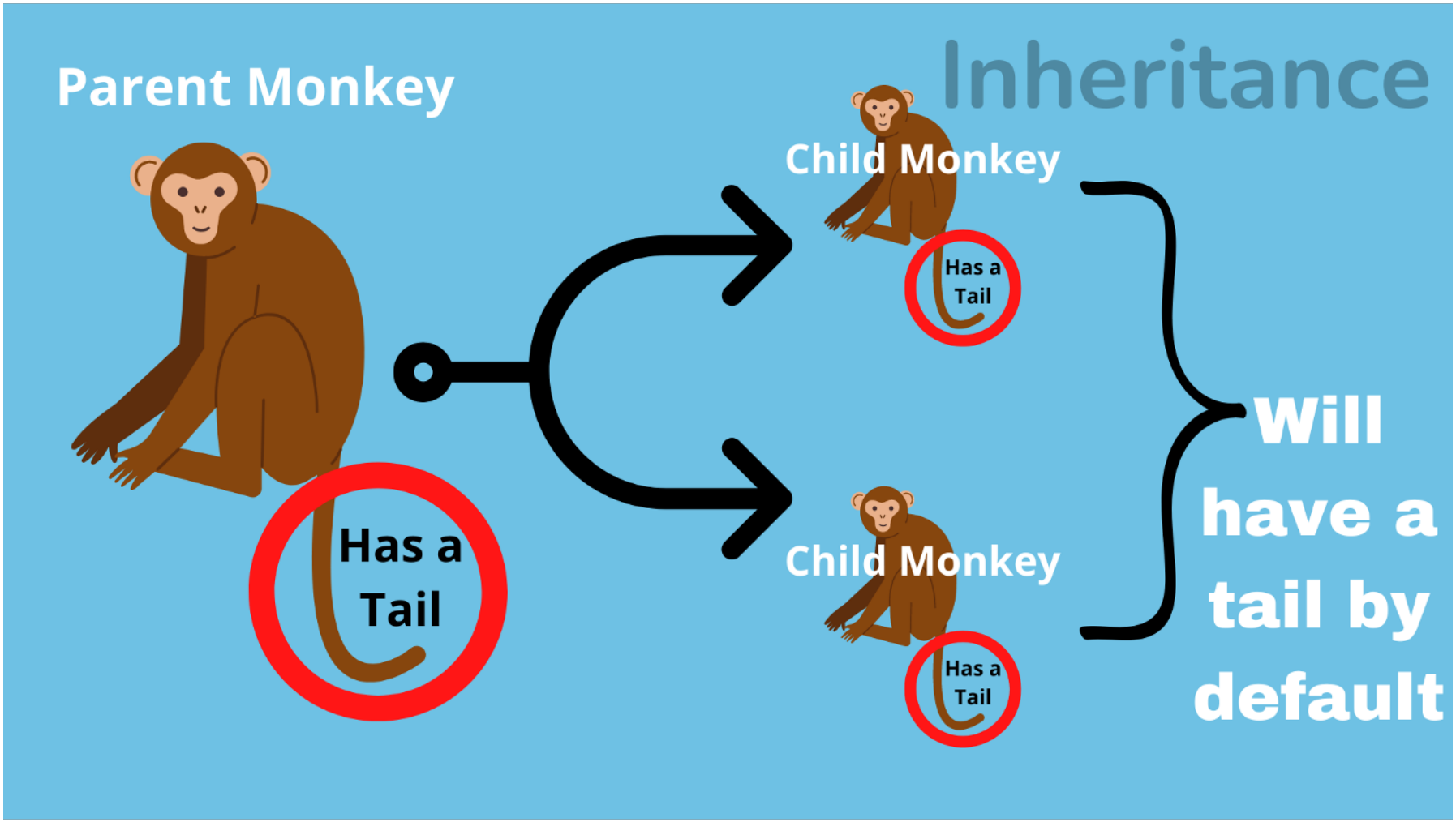
- `przedstaw_sie()` → comes from the `Zwierzak` class
- `latam()` → comes from the `Latajacy` class
- `spiewaj()` → comes from the `Ptak` class
- `pokaz_kolor()` → method specific to the `Papuga` class

```python
# Klasa bazowa 1
class Zwierzak:
    def __init__(self, imie):
        self.imie = imie

    def przedstaw_sie(self):
        print(f"Jestem {self.imie}")

# Klasa bazowa 2
class Latajacy:
    def latam(self):
        print("Potrafię latać!")

# Klasa bazowa 3
class Ptak:
    def spiewaj(self):
        print("Ćwir, ćwir!")

# Klasa Papuga dziedziczy po wszystkich trzech klasach
class Papuga(Zwierzak, Latajacy, Ptak):
    def __init__(self, imie, kolor):
        Zwierzak.__init__(self, imie)  # Wywołanie konstruktora klasy Zwierzak
        self.kolor = kolor

    def pokaz_kolor(self):
        print(f"Mam piękne {self.kolor} pióra.")

# Tworzenie obiektu klasy Papuga
papuga = Papuga("Kuba", "niebieskie")

# Wywoływanie metod odziedziczonych od różnych klas
papuga.przedstaw_sie()  # z klasy Zwierzak
papuga.latam()          # z klasy Latajacy
papuga.spiewaj()        # z klasy Ptak
papuga.pokaz_kolor()    # metoda zdefiniowana w Papuga
```

Comparison of the two approaches and inheritance in Python

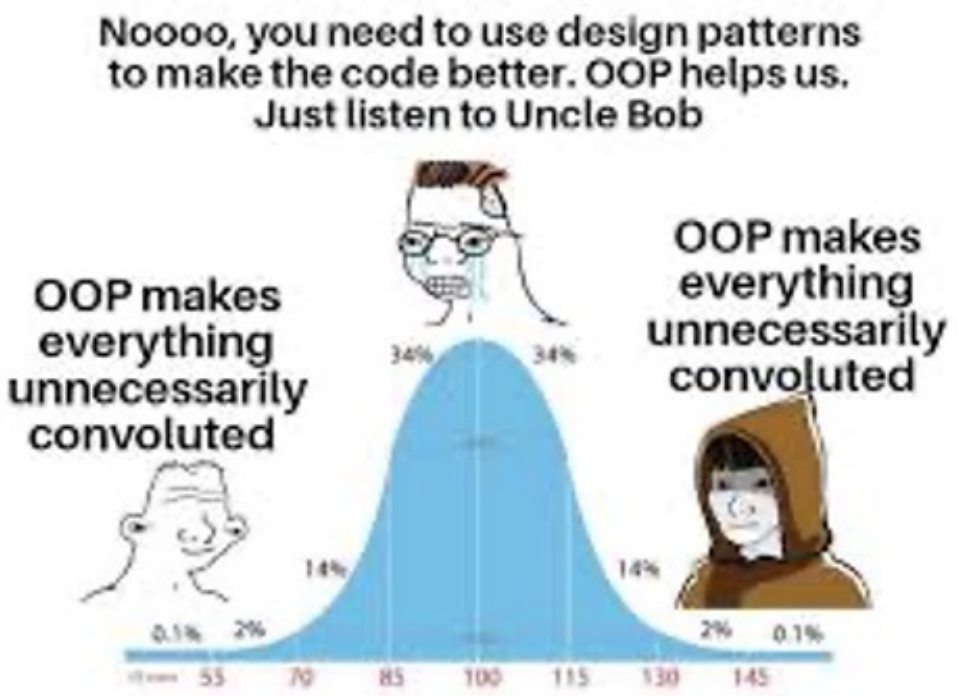
- **First code:** Repeats code, because `Cat` and `Dog` are independent and must define everything separately.
- **Second code:** Uses **inheritance**, which reduces code duplication and makes it easier to maintain.
- **Advantage:** Inheritance makes the program more **flexible** and **extensible**, because we can easily add new classes without copying code.
- **When to use inheritance?**
  - When different classes share common features and methods — it's worth creating a **base class** and inheriting from it.
  - When each class is completely unique and has no common features — it's better **not to use inheritance**.

Inheritance



Basic object-oriented programming concepts in Python

- A **class** is a **template** for objects.
- **Objects** are **instances of a class**.
- **Methods** define **object behavior**.
- **Inheritance** allows **extending class functionality**.



OOP vs FP

- **OOP (Object-Oriented Programming):** Objects maintain **state**, and methods can **modify** that state. This can lead to **side effects** when one method changes the object's state, affecting how other methods behave.
- **FP (Functional Programming):** Focuses on **immutability of data**. Functions do not change the state of the data they operate on, which leads to more **predictable** and **testable** code.

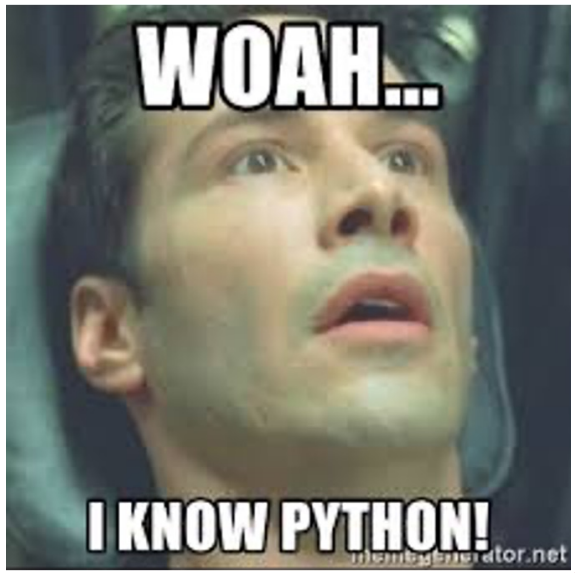
When to choose OOP and when FP?

If...	Choose FP	Choose OOP
You are creating small scripts	✓	✗
You are creating a large project	✗	✓
You have many repetitive operations on data	✓	✗
You need code organization into objects	✗	✓
You process large datasets	✓	✗
You create an application with many objects and their interactions	✗	✓

Conclusions

- **Functional Programming (FP)** → Better for working with data, more **mathematical** and **modular**. It makes it easier to create **clean**, **predictable**, and easily testable code.

- **Object-Oriented Programming (OOP)** → Better for building **large applications**, allows organizing code into **objects** and their **interactions**. It makes it easier to **extend** and **maintain** complex systems.



Supplementary materials

- Introduction to OOP in Python – DataMentor
- OOP vs Functional Programming – comprehensive comparison
- Python10MinutesADay – Parents, Children and Inheritance (GitHub)
- YouTube: Object Oriented Programming tutorial (time 25:18)
- PPS Theory – Lab 6 (Poznań University of Technology)
- YouTube: Functional vs Object Oriented Programming

OOP in meme