

# Getting Started with Object-Oriented Programming in Python

**Guest Lectures at TTK University of Applied Sciences, Tallinn, Estonia**

**8 – 12 December 2025**

**Andy Louwyck & Dominique Stove**

**Vives University of Applied Sciences, Kortrijk, Belgium**

# Who's Teaching Today?

## Andy Louwyck

- Master and Doctor in Science: Geology
- Associate Degree in Programming
- Micro Degree in AI & Data Science
- Lecturer in IT at Vives University of Applied Sciences
- AI Coordinator at Flanders Environment Agency

## Dominique Stove

- Master in Applied Economics
- Teaching Master's Degree in Economics, Mathematics, and Physics
- Lecturer in IT at Vives University of Applied Sciences
- IT Consultant in Infrastructure
- Founder and Business Owner of IqPro





# Vives Campus in the City of Kortrijk





# Informatics Program for Exchange Students



<https://www.vives.be/en/commercial-sciences-business-management-and-informatics/informatics-kortrijk>

The Informatics-programme is a programme consisting of lectures, group work, visits and projects in the field of Business and Informatics. Evaluation follows the rules of the European Credit Transfer System (ECTS). Incoming students can select a programme of up to 30 ECTS credits per semester.

**New full-year program!**

Title	ECTS	hours/week S1	hours/week S2	Semester
Introduction to Artificial Intellingence	5	3	0	1
Programming in Python	3	2	0	1
Digital Workplace	3	2	0	1
Android App Development	5	3	0	1
E-business en E-marketing	3	2	0	1
Introduction to linux	3	2	0	1
Cybersecurity	5	3	0	1
Professional and International Communication 3 (English)	3	2	0	1
	<b>30</b>	<b>19</b>	<b>0</b>	
Machine Learning - Fundamentals	6	0	4	2
IT-Project	5	0	3	2
Power Tools	3	0	3	2
Full-Stack Development in .NET	6	0	4	2
Mobile App Development iOS	5	0	4	2
Data Engineering	5	0	3	2
Node.js Development	3	0	2	2
	<b>33</b>	<b>0</b>	<b>23</b>	

# Let's Dive In!

1. Introduction: how to structure code?
2. What is object-oriented programming (OOP)?
3. From theory to code: OOP in Python
4. Demo + exercises



Spaghetti Code

Guest Lectures at TTK University of Applied Sciences, Tallinn, Estonia  
Getting Started with Object-Oriented Programming in Python

# HOW TO STRUCTURE MY CODE?

# Which code would you write?

## Approach 1

```
1 print(3.14159 * 5 * 2)
2
```

## Approach 2

```
1 def calculate_area(radius):
2     pi = 3.14159
3     area = pi * radius ** 2
4     return area
5
6 print(calculate_area(5))
```

# Which code would you write?

## Approach 1

### Pros:

- Very short and simple
- Quick for one-time use

### Cons:

- Hard to reuse
- No clear structure
- Difficult to maintain

## Approach 2

### Pros:

- Code is reusable
- Better structure
- Easier to maintain and extend
- Hides details by using local variables

### Cons:

- Slightly more code
- Requires understanding functions



# What do you think of this object-oriented version?

```
1  class Circle:
2
3      def __init__(self, radius):
4          self.radius = radius
5          self.pi = 3.14159
6
7      def area(self):
8          return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```

# What do you think of this object-oriented version?

## Pros

- Well structured
- Groups data and behavior in one place
- Keeps related details together
- Easy to reuse for many circles
- Easy to maintain
- Easy to extend (e.g. add perimeter)

## Cons

- More code than a simple function
- Adds an extra concept: classes and objects
- Slightly harder for beginners to read
- Overkill for very small scripts

# Components in this OO example

class definition

```
1 class Circle:
```

```
2
```

constructor

```
3
```

```
    def __init__(self, radius):
```

attributes

```
        self.radius = radius
```

```
        self.pi = 3.14159
```

```
4
```

```
5
```

method

```
    def area(self):
```

self refers to the created object

```
        return self.pi * (self.radius ** 2)
```

```
6
```

```
10 c = Circle(5)
```

object instantiation

```
11 print(c.area())
```

method call

Class definition — `class Circle:`

The blueprint that groups related **data** and **behavior**.

class definition

```
1  class Circle:
2
3      def __init__(self, radius):
4          self.radius = radius
5          self.pi = 3.14159
6
7      def area(self):
8          return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```



**Constructor (\_\_init\_\_)** — `def __init__(self, radius):`  
Runs when you create an object; **initializes** the object's state.

```
1 class Circle:
2     constructor
3     def __init__(self, radius):
4         self.radius = radius
5         self.pi = 3.14159
6
7     def area(self):
8         return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```

Instantiation — `c = Circle(5)`

Creates a **new object** (an instance of `Circle`).

```
1  class Circle:
2
3      def __init__(self, radius):
4          self.radius = radius
5          self.pi = 3.14159
6
7      def area(self):
8          return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```

object instantiation

## **self** (the object itself)

Refers to the **current instance**; used to access attributes and methods.

```
1 class Circle:
2
3     def __init__(self, radius):
4         self.radius = radius
5         self.pi = 3.14159
6
7     def area(self):
8         return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```


**self** refers to the created object

Attributes — `self.radius`, `self.pi`

Data stored inside each object (the object's state).

```
1  class Circle:
2
3      def __init__(self, radius):
4          self.radius = radius
5          self.pi = 3.14159
6
7      def area(self):
8          return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```

attributes





Method — `def area(self):`

Behavior that uses the object's data to compute a result.

```
1  class Circle:
2
3      def __init__(self, radius):
4          self.radius = radius
5          self.pi = 3.14159
6
7      method → def area(self):
8          return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```

## Method call (dot notation) — `c.area()`

Invokes the object's behavior; returns the computed area.

```
1  class Circle:
2
3      def __init__(self, radius):
4          self.radius = radius
5          self.pi = 3.14159
6
7      def area(self):
8          return self.pi * (self.radius ** 2)
9
10 c = Circle(5)
11 print(c.area())
```

method call

# Circles also know their perimeter!

```
1 class Circle:
2
3     def __init__(self, radius):
4         self.radius = radius
5         self.pi = 3.14159
6
7     def area(self):
8         return self.pi * (self.radius ** 2)
9
10    def perimeter(self):
11        return 2 * self.pi * self.radius
12
13 c = Circle(5)
14 print(c.area())
15 print(c.perimeter())
```

methods

method calls

# Which code do you prefer?

## Procedural programming

```
1 pi = 3.14159
2
3 def area(radius):
4     return pi * (radius ** 2)
5
6 def perimeter(radius):
7     return 2 * pi * radius
8
9 radius = 5
10 print(area(radius))
11 print(perimeter(radius))
```

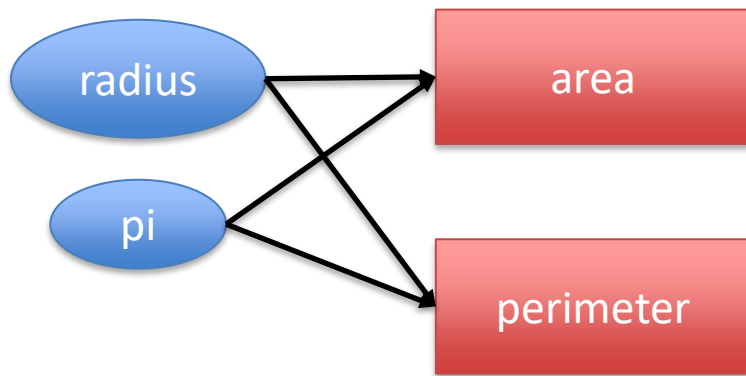
## Object-oriented programming (OOP)

```
1 class Circle:
2
3     def __init__(self, radius):
4         self.radius = radius
5         self.pi = 3.14159
6
7     def area(self):
8         return self.pi * (self.radius ** 2)
9
10    def perimeter(self):
11        return 2 * self.pi * self.radius
12
13 c = Circle(5)
14 print(c.area())
15 print(c.perimeter())
```



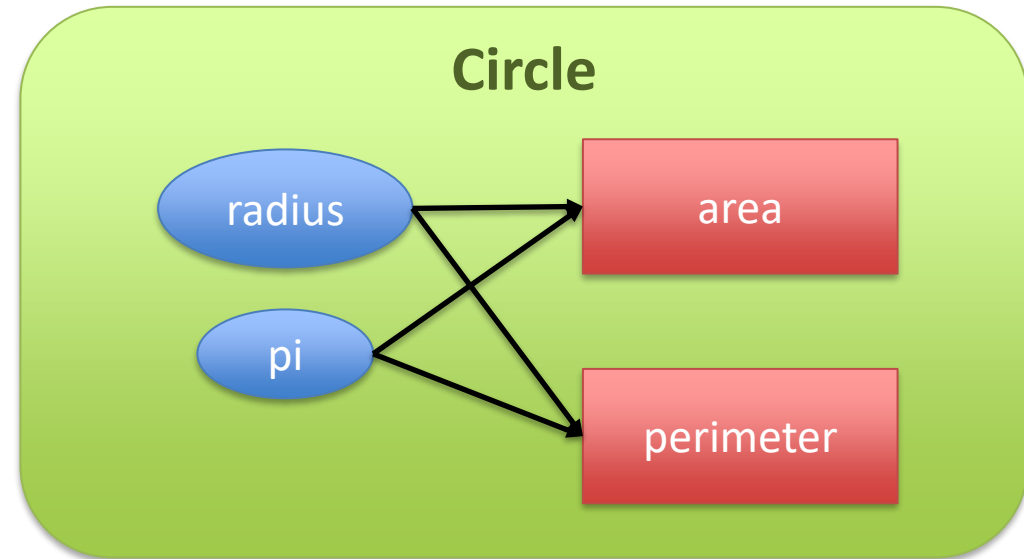
# Which code do you prefer?

## Procedural programming



Variables (data) and functions (behavior) are separate; they are **not bundled** into a single structure

## Object-oriented programming (OOP)



A **class** that **encapsulates** both variables (data) and functions (behavior) in one single structure

# Which code do you prefer?

## Procedural programming

### Pros:

- Simple for small scripts
- Easy to understand for beginners
- Quick to implement

### Cons:

- Data and behavior are separate
- Harder to maintain when many related functions
- No clear structure for scaling

## Object-oriented programming (OOP)

### Pros:

- Groups data and behavior together
- Easier to reuse and extend
- Better structure for complex systems

### Cons:

- More code for simple tasks
- Slightly more overhead
- Requires understanding of classes and objects

# There is a third approach...

Besides procedural and object-oriented programming, Python also supports **functional programming** (which is beyond the scope of this lecture):

```
1 import math
2
3 radii = [2, 3.5, 5]
4
5 areas = list(map(lambda r: math.pi * (r ** 2), radii))
6 perimeters = list(map(lambda r: 2 * math.pi * r, radii))
7
8 print(areas)
9 print(perimeters)
```

*“No state, just transform a list of radii into a list of areas/perimeters”*

Guest Lectures at TTK University of Applied Sciences, Tallinn, Estonia  
Getting Started with Object-Oriented Programming in Python

# WHAT IS OBJECT-ORIENTED PROGRAMMING?

# Classes and Objects



## Class

- A **blueprint** for creating objects
- Defines **attributes** and **methods**
- Example:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14159 * (self.radius ** 2)
```

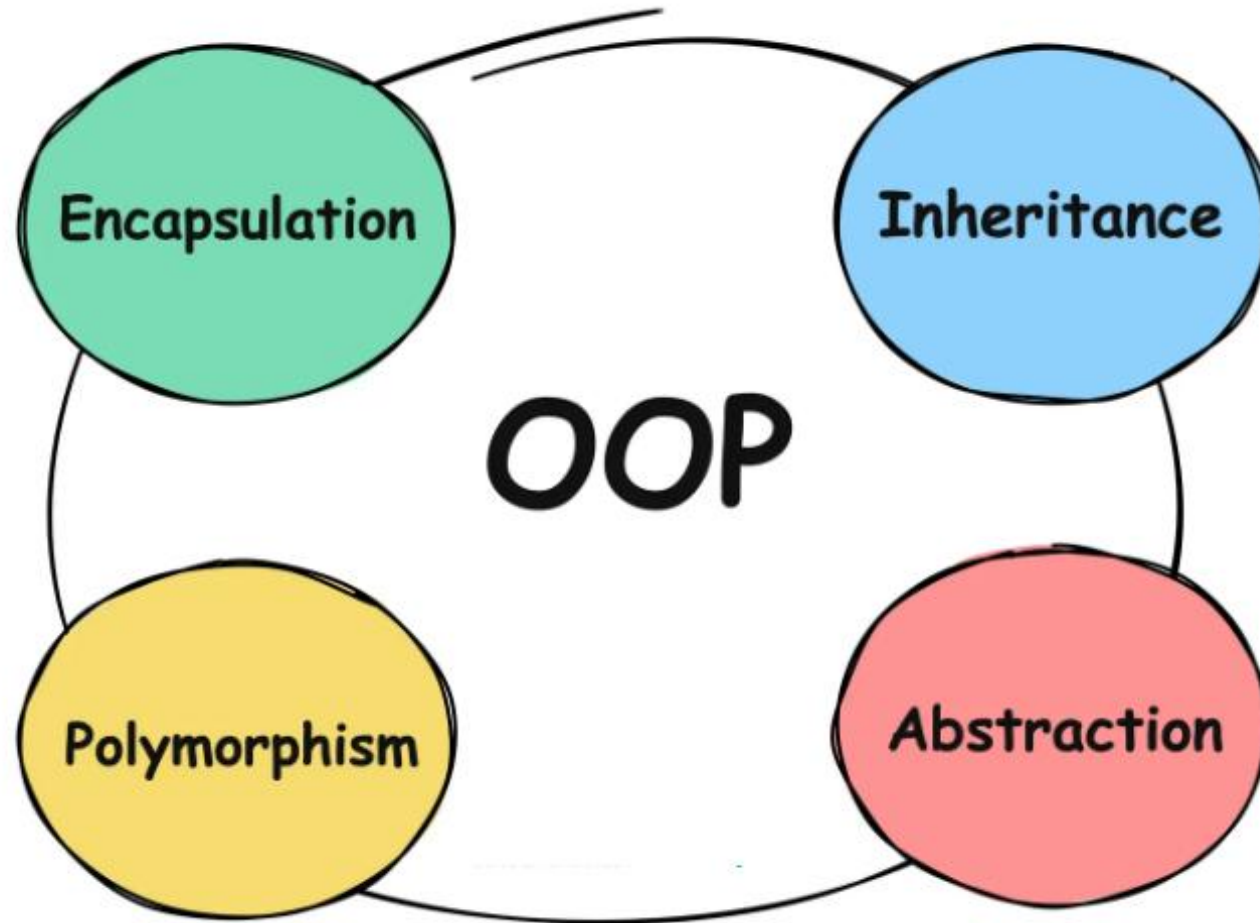
## Object

- An **instance** of a class
- Has its own **state** (= attribute values)
- Example:

```
c = Circle(5)
print(c.area())
```

*“c is an object of class **Circle** and its state is defined by a radius of 5”*

# The 4 Main Features of Object-Oriented Programming



# The 4 Main Features of Object-Oriented Programming

## 1. Encapsulation

- **Bundling data** (attributes) **and behavior** (methods) inside a single unit (class).
- Controls access to internal details, exposing only what's necessary.

## 2. Inheritance

- Creating new classes based on existing ones, **reusing and extending functionality**.
- Example: the circle class could inherit from a cylinder class and adopt the area method

## 3. Polymorphism

- Same interface, **different implementations**.
- Example: the area method is implemented for circle and cylinder classes each with its own logic.

## 4. Abstraction

- Hiding implementation details and showing only the **essential interface**.
- Example: you call the area method from a circle object without worrying about the formula



# Encapsulation

## Definition:

Encapsulation means bundling data and methods inside a class and restricting direct access to some attributes to protect the integrity of the data.

## Example:

```
1 class BankAccount:
2     def __init__(self, balance):
3         self.__balance = balance # private attribute
4
5     def deposit(self, amount):
6         self.__balance += amount
7
8     def get_balance(self):
9         return self.__balance
10
11 account = BankAccount(100)
12 account.deposit(50)
13 print(account.get_balance()) # Output: 150
```

# Inheritance

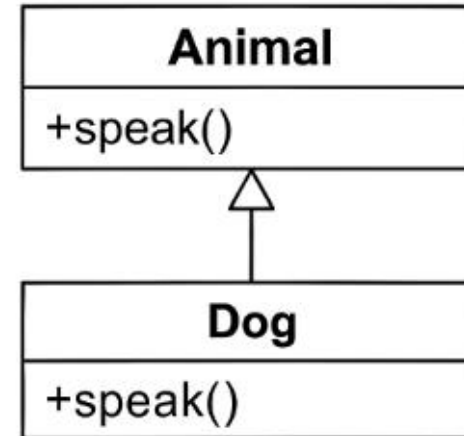
## Definition:

Inheritance allows a class to acquire properties and methods from another class, promoting code reuse.

## Example:

```
1 class Animal:
2     def speak(self):
3         print("I am an animal")
4
5 class Dog(Animal):
6     def speak(self):
7         print("Woof!")
8
9 dog = Dog()
10 dog.speak() # Output: Woof!
```

## UML Class Diagram



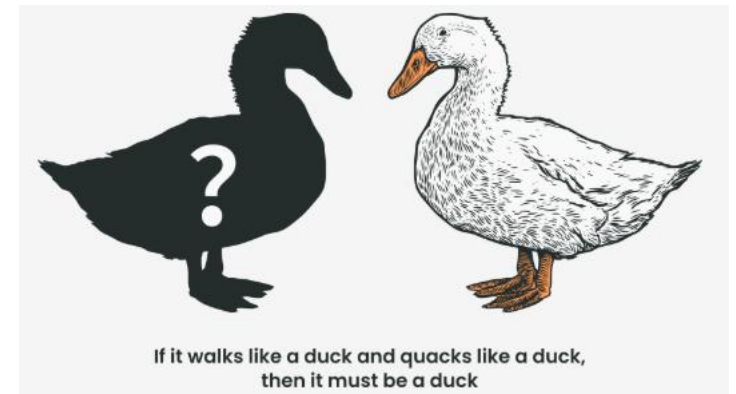
# Polymorphism

## Definition:

Polymorphism means the same method name can behave differently depending on the object.

## Example:

```
1 class Cat:
2     def speak(self):
3         return "Meow"
4
5 class Dog:
6     def speak(self):
7         return "Woof"
8
9 def animal_sound(animal):
10     print(animal.speak())
11
12 animal_sound(Cat()) # Output: Meow
13 animal_sound(Dog()) # Output: Woof
```



# Abstraction

## Definition:

Abstraction hides complex implementation details and exposes only the essential features. This is often done using **abstract base classes** (which is out of scope).

## Example:

```
1  from abc import ABC, abstractmethod
2
3  class Shape(ABC):
4      @abstractmethod
5      def area(self):
6          pass
7
8  class Circle(Shape):
9      def __init__(self, radius):
10         self.radius = radius
11
12         def area(self):
13             return 3.14 * self.radius ** 2
14
15  circle = Circle(5)
16  print(circle.area()) # Output: 78.5
```

# Why Use Object-Oriented Programming?

- **Why OOP?**
  - Organizes code into **classes and objects**, making it easier to manage.
  - Models real-world entities, improving **readability and understanding**.
- **Key Benefits:**
  - **Reusability:** Use existing classes through inheritance.
  - **Maintainability:** Easier to update and extend code.
  - **Encapsulation:** Keeps data and behavior together, reducing complexity.
  - **Scalability:** Ideal for large applications with many interacting components.

# Inheritance In Python

Guest Lectures at TTK University of Applied Sciences, Tallinn, Estonia  
Getting Started with Object-Oriented Programming in Python

## FROM THEORY TO CODE: OOP IN PYTHON

Base Class

Drived Class

# Defining Classes and Creating Objects

## Explanation:

In Python, classes are created using the `class` keyword.

Objects are instances of classes.

## Code example:

```
1  # Define a class
2  class Car:
3      def __init__(self, brand, model):
4          self.brand = brand
5          self.model = model
6
7  # Create an object
8  my_car = Car("Tesla", "Model 3")
9  print(my_car.brand)  # Output: Tesla
```



# Constructor and Attributes

## Explanation:

The `__init__` method acts as a constructor and initializes object attributes when an instance is created.

## Code example:

```
1 class Student:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6 student = Student("Alice", 21)
7 print(student.name, student.age)  # Output: Alice 21
```

# Methods in Python Classes

## Explanation:

Methods are functions that belong to a class and are defined using the `def` keyword.

- **Instance methods:** Operate on object attributes and require `self`
- **Static methods:** Don't require `self` as they don't need object attributes; use `@staticmethod`

## Code example:

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5     def greet(self): # instance method
6         return "Hello, my name is " + self.name
7
8     @staticmethod
9     def info(): # static method
10        return "I am a person."
11
12 # Usage
13 p = Person("Alice")
14 print(p.greet())      # Output: Hello, my name is Alice
15 print(Person.info())  # Output: I am a person.
```

# Access Modifiers

## Explanation:

Python uses naming conventions for access control:

- **Public:** `self.attribute`
- **Private:** `self.__attribute`

## Code example:

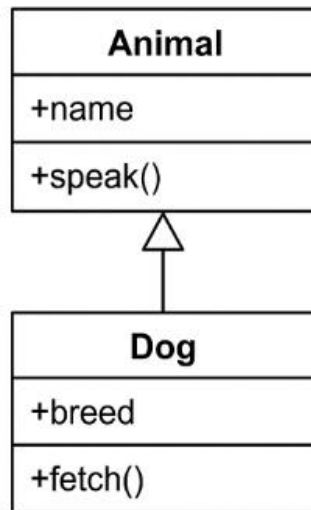
```
1 class BankAccount:
2     def __init__(self, owner, balance):
3         self.owner = owner          # public
4         self.__balance = balance    # private
5
6     def deposit(self, amount):
7         self.__balance += amount
8
9     # Usage
10    account = BankAccount("Alice", 100)
11    print(account.owner)              # OK
12    # print(account.__balance)        # Error: Attribute is private
13    account.deposit(50)
```

# Inheritance

## Explanation:

Inheritance allows a subclass to reuse attributes and methods from a superclass. The subclass can **add new attributes or methods** beyond what is inherited.

## Code example:



```
1  # Parent class
2  class Animal:
3      def __init__(self, name):
4          self.name = name
5
6      def speak(self):
7          print(self.name + " makes a sound.")
8
9  # Child class (inherits from Animal)
10 class Dog(Animal):
11     def __init__(self, name, breed):
12         super().__init__(name) # Call parent constructor
13         self.breed = breed     # New attribute
14
15     def fetch(self):
16         print(self.name + " is fetching the ball!")
17         print("Breed: " + self.breed)
18
19 # Usage
20 dog = Dog("Buddy", "Golden Retriever")
21 dog.speak() # Inherited method
22 dog.fetch() # New method
```

- Use `class Child(Parent):` to inherit.
- `super().__init__()` calls the parent constructor.
- Subclass can add **extra attributes and methods**.

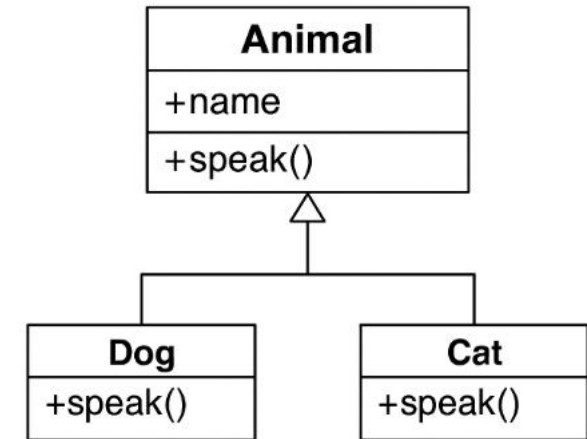
# More than one subclass

## Explanation:

A superclass can have **multiple subclasses**, each of which can have its own subclasses.

## Code example:

```
1  # Parent class
2  class Animal:
3      def __init__(self, name):
4          self.name = name
5
6      def speak(self):
7          print(self.name + " makes a sound.")
8
9  # First child class
10 class Dog(Animal):
11     def speak(self):
12         print(self.name + " barks!")
13
14 # Second child class
15 class Cat(Animal):
16     def speak(self):
17         print(self.name + " meows!")
18
19 # Usage
20 dog = Dog("Buddy"); dog.speak()  # Buddy barks!
21 cat = Cat("Whiskers"); cat.speak()  # Whiskers meows!
```



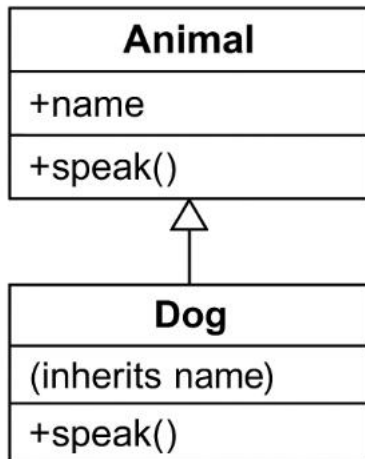
Subclasses override the `speak` method

# Method overriding

## Explanation:

A subclass can **redefine a method** from its parent class. The new method replaces the inherited one when called on the child object.

## Code example:



```
1  # Parent class
2  class Animal:
3      def __init__(self, name):
4          self.name = name
5
6      def speak(self):
7          print(self.name + " makes a sound.")
8
9  # Child class overrides speak()
10 class Dog(Animal):
11     def speak(self):
12         super().speak() # Call parent method
13         print(self.name + " barks!") # Custom behavior
14
15 # Usage
16 animal = Animal("Generic Animal"); animal.speak() # Generic Animal makes a sound.
17 dog = Dog("Buddy"); dog.speak() # Buddy makes a sound. Buddy barks!
```

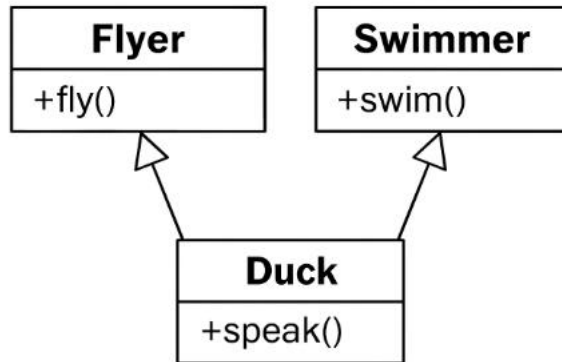
- self refers to the current object instance.
- super().method() calls the parent's implementation.
- Overriding allows **custom behavior** for subclasses.

# Multiple inheritance

## Explanation:

A class can inherit from **more than one parent class**. This allows combining functionality from multiple sources. But use with care as it can lead to complexity!

## Code example:



```
1  # First parent class
2  class Flyer:
3      def fly(self):
4          print("I can fly!")
5
6  # Second parent class
7  class Swimmer:
8      def swim(self):
9          print("I can swim!")
10
11 # Child class inherits from both
12 class Duck(Flyer, Swimmer):
13     def speak(self):
14         print("Quack!")
15
16 # Usage
17 duck = Duck()
18 duck.fly()    # I can fly!
19 duck.swim()   # I can swim!
20 duck.speak()  # Quack!
```

- Syntax: `class Child(Parent1, Parent2):`
- Methods from both parents are available in the child.

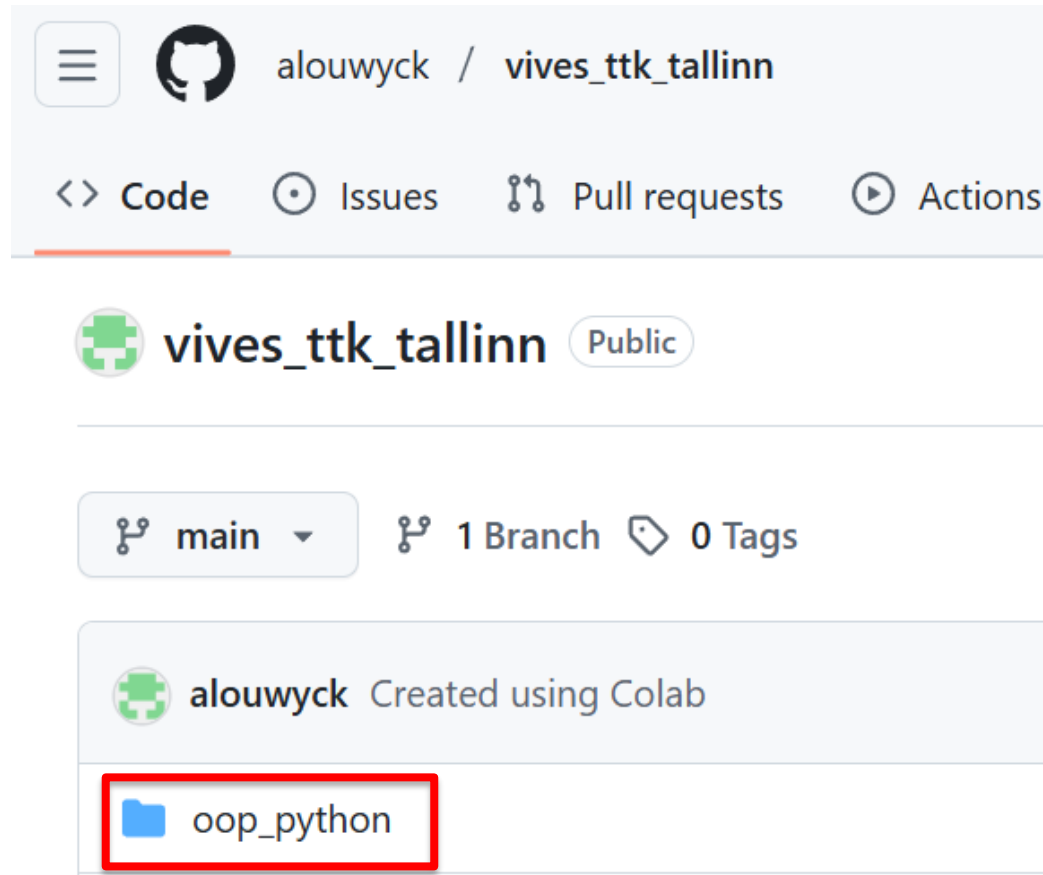


Guest Lectures at TTK University of Applied Sciences, Tallinn, Estonia  
Getting Started with Object-Oriented Programming in Python

# **OOP IN PYTHON: EXERCISES**

# GitHub Repo

[https://github.com/alouwyck/vives\\_ttk\\_tallinn](https://github.com/alouwyck/vives_ttk_tallinn)



The screenshot shows the GitHub interface for the repository 'vives\_ttk\_tallinn' by user 'alouwyck'. The repository is public. The 'Code' tab is selected. Below the repository name, it shows '1 Branch' (main) and '0 Tags'. A commit by 'alouwyck' is shown, created using Colab. A folder named 'oop\_python' is highlighted with a red box.

alouwyck / vives\_ttk\_tallinn

Code Issues Pull requests Actions

vives\_ttk\_tallinn Public

main 1 Branch 0 Tags

alouwyck Created using Colab

oop\_python

# Sources

- <https://realpython.com/python3-object-oriented-programming/>
- <https://www.geeksforgeeks.org/python/python-oops-concepts/>
- [https://www.w3schools.com/python/python\\_oop.asp](https://www.w3schools.com/python/python_oop.asp)
- <https://blog.algomaster.io/p/basic-oop-concepts-explained-with-code>
- <https://www.geeksforgeeks.org/python/python-classes-and-objects>
- <https://www.cspsprotocol.com/python-inheritance-tutorial/>
- Microsoft Copilot using GPT-5 was applied to generate some of the content