Python Level 3

Advanced Course
Class 2

Catarina Sousa Santos

October 9th, 2025

AGENDA

Part I – Object Oriented Programming

1. **Properties**
    a. Inheritance
    b. Encapsulation
    c. Abstraction
    d. Polymorphism
    e. Modularity

OBJECT-ORIENTED PROGRAMMING

PROPERTIES

OOP PROPERTIES

INHERITANCE

- Allows a class to **inherit attributes and methods from another class**. The class being inherited from is called the **base class or parent class**, and the class that inherits is called the **derived class or child class.**
- Inheritance **enables the creation of a new class that has the same behavior** as the existing class, with the possibility to add new behaviors or modify existing ones.

Motivation.txt

```
class Account:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

# Inheriting from account
class SavingsAccount(Account):
    def __init__(self, balance=0, interest_rate=0.01):
        super().__init__(balance)
        self.interest_rate = interest_rate


    def add_interest(self):
        interest = self.balance * self.interest_rate
        self.deposit(interest)  # Call method of the parent class
# Usage
savings = SavingsAccount(balance=1000)
savings.add_interest()
savings.deposit(500)
print(savings.balance) # 1510.0
```

OOP – INHERITANCE

INHERITANCE

- The **SavingsAccount** class **inherits** from Account, meaning it has access to the balance attribute and inherits the deposit method.
- **SavingsAccount** uses **super().__init__(balance)** to **call the constructor of Account** to initialize the balance attribute.
- **SavingsAccount** adds a new attribute **interest_rate,** specific to this subclass.
- The **add_interest** method in **SavingsAccount** calculates the interest based on the current balance and the **interest_rate**. It then uses the **inherited deposit method** to add this interest to the balance.

Motivation.txt

```
class Account:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

# Inheriting from account
class SavingsAccount(Account):
    def __init__(self, balance=0, interest_rate=0.01):
        super().__init__(balance)
        self.interest_rate = interest_rate


    def add_interest(self):
        interest = self.balance * self.interest_rate
        self.deposit(interest)  # Call method of the parent class
# Usage
savings = SavingsAccount(balance=1000)
savings.add_interest()
savings.deposit(500)
print(savings.balance) # 1510.0
```

OOP – ENCAPSULATION

ENCAPSULATION

- The **Account** class has an attribute balance, which is set to **public**. This implies there's **no restriction on its access**. The class provides the *deposit* method to allow controlled modification of this attribute, be it from **inside** or **outside** the class.
- Although *balance* is public, by using the *deposit* and *add_interest* methods, the class directs how balance should be modified. This is a form of **encapsulation:** the object's state is modified through a **controlled interface**, not **arbitrarily from outside the class.**
- However, the attributes can still be modified directly from outside the class.

Motivation.txt

```
class Account:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

# Inheriting from account
class SavingsAccount(Account):
    def __init__(self, balance=0, interest_rate=0.01):
        super().__init__(balance)
        self.interest_rate = interest_rate


    def add_interest(self):
        interest = self.balance * self.interest_rate
        self.deposit(interest)  # Call method of the parent class
# Usage
savings = SavingsAccount(balance=1000)
savings.add_interest()
savings.deposit(500)
print(savings.balance) # 1510.0
```

OOP – ENCAPSULATION

ENCAPSULATION

Encapsulation involves **bundling the attributes and methods** that operate on the data into a **single unit or class** and **restricting access** to some of the object's components.

It doesn't just mean **hiding data**, but also providing **controlled interfaces** (methods) to **manage the object's state**. Three types of attributes can be defined within a class:

- **Public attributes**: These are **accessible from anywhere**, both **inside** and **outside** of the class. There is **no restriction** on access, which means that any code that creates an object of the class can directly access and modify public attributes.
- **Protected attributes**: These are **accessible within the class and by subclasses** of the class. They are meant to warn users of the class that such variables are internal to the class hierarchy and **should not be accessed directly**. Unlike private variables, protected variables can be accessed in subclasses, making them less restrictive.
- **Private attributes**: These are meant to be **completely hidden from any external code**. Private variables can **only be accessed within the class that defines them**. This is the strictest level of encapsulation.

OOP – ENCAPSULATION

ENCAPSULATION

Restricting access to some of the object's components is usually done to hide the internal representation, or state, of the object from the outside.

In Python, encapsulation is achieved using **private** and **protected** members, denoted by underscores before the member names.

Private: __

Protected: _

Motivation.txt

```
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance # private
        self._account_type = "Checking"


    def deposit(self, amount):
        if amount >= 0:
            self.__balance += amount
            print(f"Added {amount} to the balance")
        else:
            print("Deposit amount must be positive")


    def withdraw(self,amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount} from the balance")
        else:
            print("Insufficient balance or invalid withdrawal amount")


    def get_balance(self):
        return self.__balance

    def get_account_type(self):
        return self._account_type

acc = Account("José Silva", 20000)
print(acc.__balance) # AttributeError: 'Account' object has no attribute '__balance'
print(acc.get_balance()) # 20 000
print(acc._account_type) # Checking
print(acc.get_account_type()) # Checking
```

OOP – ENCAPSULATION

PROTECTED ATTRIBUTES

- **_account_type** is a **protected** attribute in SavingsAccount inherited from the Account class.
- SavingsAccount **can access and modify** the protected _account_type attribute directly as shown in __init__ and get_account_details methods
- This is consistent with the idea that **subclasses should be able to access protected attributes, but external entities should not**

Motivation.txt

```
# Subclassing to demonstrate protected variable access
class SavingsAccount(Account):
    def __init__(self, owner, balance=0):
        super().__init__(owner, balance)
        # modifying protected attribute
        self._account_type = "Savings"


    # Accessing protected variable from a subclass
    def get_account_details(self):
        return f"Account Type: {self._account_type}, Balance:
{self.get_balance()}"


savings_acc = SavingsAccount("Anna Smith")
savings_acc.get_account_details() # 'Account Type: Savings,
                                                    Balance: 0'
```

OOP – ENCAPSULATION

PROTECTED ATTRIBUTES

In Python, the concept of protected attributes is **more about convention than enforcement**. Protected attributes are meant to be a signal to other programmers that these variables are **intended for internal use within the class and its subclasses**, but this is not strictly enforced by the Python runtime.

The *_account_type* attribute is protected, and while it's intended for use within the class and subclasses, you can technically still access and modify it from outside the class due to the conventions of Python.

Motivation.txt

```
account = SavingsAccount("John Doe", 1000)
print(account.get_account_type())
# Prints "Savings", which is set in the
subclass constructor


# Inappropriately modifying the protected
attribute from outside the class
account._account_type = "Checking"
print(account.get_account_type())
# Prints "Checking", reflecting the
external modification
```

OOP – ENCAPSULATION

PRIVATE ATTRIBUTES

The **public** methods ***deposit*** and ***withdraw*** allow interaction with the **private** ***__balance*** attribute in controlled ways. This ensures that the **balance cannot be modified arbitrarily**; it can only be changed through these methods that include checks and validations.

The method ***get_balance*** provides a way to access the private ***__balance*** attribute's value without exposing it for direct modification.

This class structure ensures that the ***__balance*** attribute is encapsulated within the Account class, and **it can only be accessed or modified through the methods provided**

Motivation.txt

```
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance # private
        self._account_type = "Checking"

    def deposit(self, amount):
        if amount >= 0:
            self.__balance += amount
            print(f"Added {amount} to the balance")
        else:
            print("Deposit amount must be positive")

    def withdraw(self,amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount} from the
balance")
        else:
            print("Insufficient balance or invalid
withdrawal amount")

    def get_balance(self):
        return self.__balance

    def get_account_type(self):
        return self._account_type
```

OOP – ENCAPSULATION

PRIVATE ATTRIBUTES

The Account class encapsulates the balance of the account (***__balance***) by making it a private attribute. This means that **it cannot be accessed directly from outside the class**.

Public methods (***deposit***, ***withdraw***, and ***get_balance***) are provided to interact with the balance in a **controlled** manner. This ensures that the balance cannot be directly modified from outside the class, **protecting it from unauthorized changes and enforcing any rules for its modification** (like preventing withdrawal of more money than the account holds).

Motivation.txt

```
# Creating an account object
acc = Account("John", 100)
# Accessing the public methods
acc.deposit(50)
acc.withdraw(20)
print(acc.get_balance()) # Output: 130


# Attempting to access the private
attribute directly from outside the class
will result in an AttributeError
print(acc.__balance()) # This will raise
an AttributeError
```

OOP – ABSTRACTION

ABSTRACTION

- Abstraction is the concept of **hiding the complex reality while exposing only the necessary parts**. It means representing essential features without including the background details or explanations
- Classes use the concept of abstraction to define and group data and methods that act on the data, and they expose only the necessary and relevant parts of their functionality
- **Abstraction** can be achieved in several ways, including using **abstract classes** and **methods**:
  - An abstract class is a class that **cannot be instantiated on its own** and is **designed to be subclassed**
  - An abstract class often includes one or more **abstract methods**, which are methods **declared in the abstract class but must be implemented by the subclass**
  - Subclasses provide specific **implementations of abstract methods**, hiding their complex details behind a simple method call

OOP – ABSTRACTION

- Account class is an **abstract class (ABC)** that defines a **common interface** for all accounts, including the *withdraw* method, which allows to modify the account balance.
- .The ***calculate_interest*** method is **declared as an abstract method using the @abstractmethod decorator**. This forces any subclass of Account to implement this method, ensuring that each type of account handles interest calculation in a way that's appropriate for that account type.
- The **SavingsAccount** subclass provides a **specific implementation** of the ***calculate_interest*** method that calculates interest based on the **account balance** and a specified **interest rate**.

Motivation.txt

```
from abc import ABC, abstractmethod

class Account(ABC):
    def __init__(self, balance=0):
        self.balance = balance # private

    def withdraw(self,amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew {amount} from the balance")
        else:
            print("Insufficient balance or invalid withdrawal amount")

    @abstractmethod
    def calculate_interest(self):
        pass

class SavingsAccount(Account):
    def __init__(self, balance=0, interest_rate=0.01):
        super().__init__(balance)
        self.interest_rate = interest_rate

    def calculate_interest(self):
        return self.balance * self.interest_rate

# Usage:
savings_account = SavingsAccount(1000, 0.02)
interest = savings_account.calculate_interest()
print(f"Interest: {interest}")
# It is not possible to instantiate abstract classes
account = Account(100) # This will raise a TypeError
```

OOP – POLIMORPHISM

POLIMORPHISM

- **Polymorphism** refers to the ability of **different objects to respond in a unique way to the same method call**
- Polymorphism allows methods to do different things based on the object it is acting upon
- Using the **Account**, **SavingsAccount**, and **CheckingAccount** classes, we can demonstrate polymorphism through the ***calculate_interest*** method. Each subclass (SavingsAccount and CheckingAccount) provides its own implementation of this method, **tailored to its specific needs**

Motivation.txt

```
from abc import ABC, abstractmethod

class Account(ABC):
    def __init__(self, balance=0):
        self.balance = balance # private

    def withdraw(self,amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew {amount} from the balance")
        else:
            print("Insufficient balance or invalid
withdrawal amount")
    @abstractmethod
    def calculate_interest(self):
        pass

class SavingsAccount(Account):
    def __init__(self, balance=0, interest_rate=0.01):
        super().__init__(balance)
        self.interest_rate = interest_rate

    def calculate_interest(self):
        return self.balance * self.interest_rate

class CheckingAccount(Account):
    def __init__(self, balance=0):
        super().__init__(balance)

    def calculate_interest(self):
        return 0
```

OOP – POLIMORPHISM

POLIMORPHISM

- By calling ***display_interest*** with either a **SavingsAccount** or **CheckingAccount** object, we see **polymorphic behavior** in action
- Even though we're calling the same method name, ***calculate_interest***, on both savings and checking, the outcome is different because of how each subclass has **overridden** the method to suit its purposes. **SavingsAccount** calcu-lates interest based on the balance and an interest rate, whereas **CheckingAccount** simply returns 0, reflecting that checking accounts might not accrue interest.
- This **ability to interact with objects of different classes** through a common interface, while still allowing each class to perform its own specific implementation of a method, is the essence of polymorphism.

Motivation.txt

```
def display_interest(account):
    print(f"Interest: {account.calculate_interest()}")

savings = SavingsAccount(balance=1000, interest_rate=0.05)
checking = CheckingAccount(balance=1000)

# This will use the SavingsAccount implementation of
calculate_interest
display_interest(savings)  # 50

# This will use the CheckingAccount implementation of
calculate_interest
display_interest(checking)  # 0
```

OOP – MODULARITY

MODULARITY

Modularity refers to the principle of **breaking down a program** into separate, independent modules, where each module handles a specific part of the program's functionality.

Separation of Concerns

- The **Account** class defines the basic framework for all types of bank accounts.
- **SavingsAccount** and **Checking-Account** extend Account, each adding functionalities and properties specific to their type.

Reusability

- The **Account** class can be reused to create any number of account types without rewriting the common functionality.
- Methods like **deposit** and **withdraw** are defined once in the Account class and reused by all subclasses, ensuring consistency and reducing redundancy.

Motivation.txt

```
class Account:
     # Common functionalities for all
accounts
    ...

class SavingsAccount(Account):
    # Specific functionalities for savings
accounts, like interest calculatio
    ...

class CheckingAccount(Account):
# Specific functionalities for checking
accounts, potentially different from
savings accounts
    ...
```

OOP – MODULARITY

MODULARITY

Maintainability

- Changes to the banking system's core logic, such as how transactions are logged, need to be updated in only one place (the **Account** class), and all account types inherit the changes
- Specific behaviors of account types can be modified independently, without affecting others. For instance, changing how interest is calculated for **SavingsAccount** doesn't impact **CheckingAccount**

Scalability

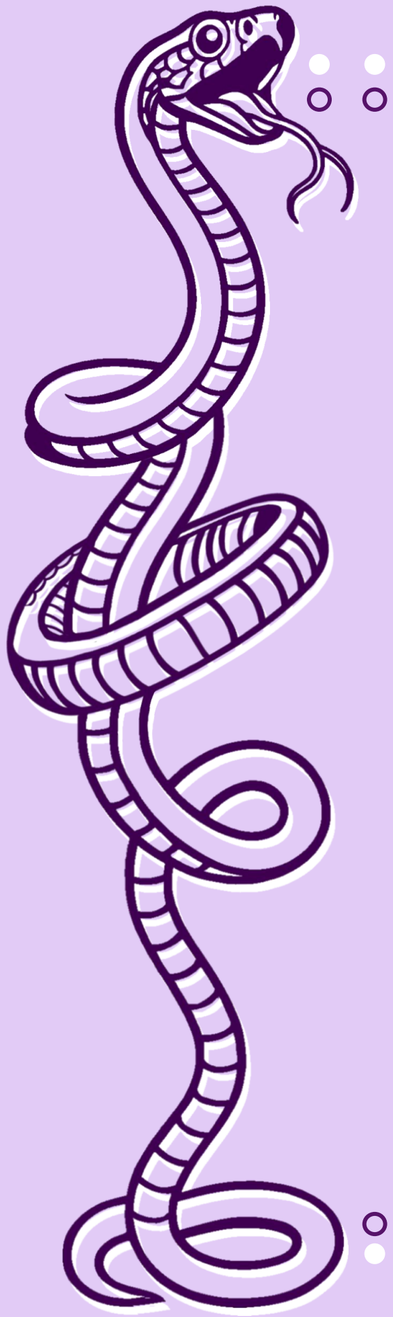
- The application can grow to accommodate new types of accounts and functionalities without major overhauls to the existing codebase. Adding a new account type involves creating a new subclass rather than modifying a monolithic structure

Motivation.txt

```
class Account:
     # Common functionalities for all
accounts
    ...

class SavingsAccount(Account):
    # Specific functionalities for savings
accounts, like interest calculatio
    ...

class CheckingAccount(Account):
# Specific functionalities for checking
accounts, potentially different from
savings accounts
    ...
```



CLASS WRAP-UP

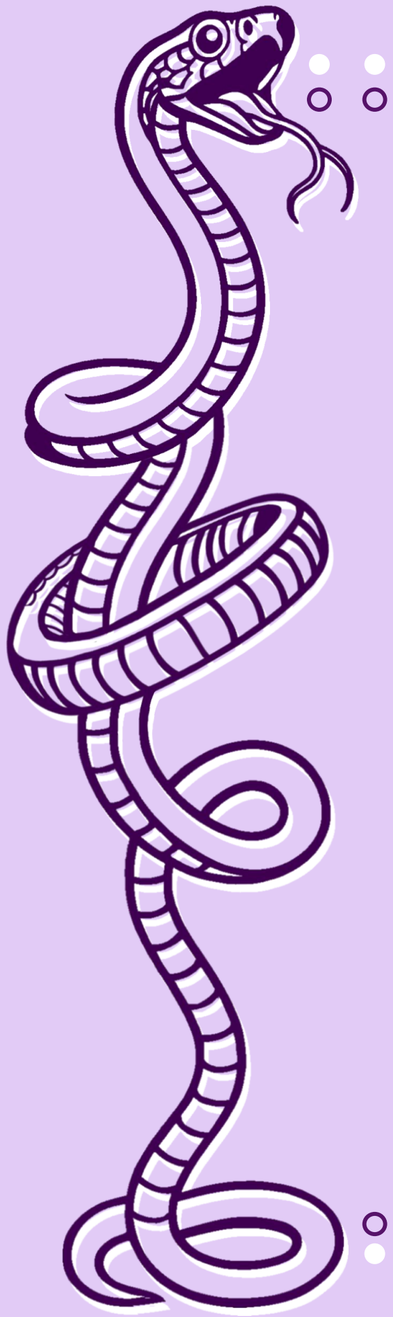
WHAT DID WE LEARN TODAY?

We've learned that **OOP** helps us **organize and structure code** around real-world objects and their interactions.

In particular, we explored the five main **OOP principles**:

- **Inheritance** allows **reusing and extending existing code by creating subclasses** that build on parent classes.
- **Encapsulation** helps **protect and manage data through controlled access** to attributes and methods.
- **Abstraction** allows us to focus on what matters by **hiding unnecessary implementation details**.
- **Polymorphism** enables **different objects to respond uniquely to the same method** call.
- **Modularity** encourages to **split programs into smaller, manageable parts**, improving clarity and maintenance.

Together, these principles make **code clearer, safer, and easier to scale and reuse**.



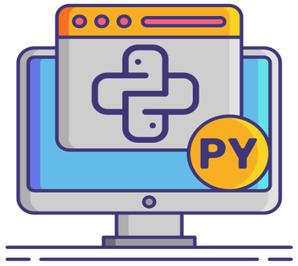
PRACTICE
PRACTICE
PRACTICE

YOU WON'T MASTER A SKILL IF YOU DON'T PRACTICE!

EXERCISES - LEARN BY DOING

In order to facilitate the learning process of Python **we have prepared for each session a python file** where you can find **exercises** that will help you to grasp the introduced Python concepts.

Visual Studio Code



We will use **VS CODE** as our Python program IDE

EXERCISES FOR TODAY

README.md — added initial course structure and .README~ — 17 hours ago

README

🐍 Natixis Python Level 3

📚 Course Overview

This advanced Python course is designed for students who have already mastered intermediate programming concepts and want to expand their skills toward object-oriented programming (OOP), modular design, and interactive application development. Students will learn to structure Python code professionally, understand reusability and abstraction, manage projects using Git and GitHub, and create Streamlit applications for data-driven interfaces.

📋 Course Structure (Intermediate Python - Level 3)

0 forks

Report repository

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

Link to exercises: https://github.com/cat-fss/natixis_python_level_3/blob/master/exercises/Class1_exercises.py

WHY SHOULD YOU DEACTIVATE CO-PILOT?

As **beginners in Python programming**, it's crucial to focus on truly understanding how code works, rather than just seeing it appear. Tools like GitHub Copilot can be tempting, but they **often offer solutions without explanation**, making it easy to skip the learning process. While these tools are designed to assist, **not replace your thinking**, they can encourage you to rely on solutions you don't fully grasp—and they're not always correct. To truly learn, you need to write, debug, and explore code on your own. **By turning off Copilot** during the early stages of your learning, you give yourself the opportunity to develop real problem-solving skills, build confidence, and create a strong foundation. Later, when you have a solid grasp of the basics, Copilot can serve as a useful support tool, but always approach its suggestions with a critical mindset, not blind trust.

Steps to turn-off GitHub Copilot:

1. Go to Settings (File > Preferences > Settings or press Ctrl+,).
2. In the search bar, type: Copilot.
3. Find the setting GitHub Copilot: Enable.
4. Uncheck it to disable Copilot globally.

THANK YOU

Questions?

Catarina Santos catfssantos@gmail.com