Python

Object Oriented Programming

JUNIA ISEN / M1 / 2024-2025
Nacim Ihaddadene

First, let’s disable copilot!

And ChatGPT

And Gemini

And Mistral

And Claude

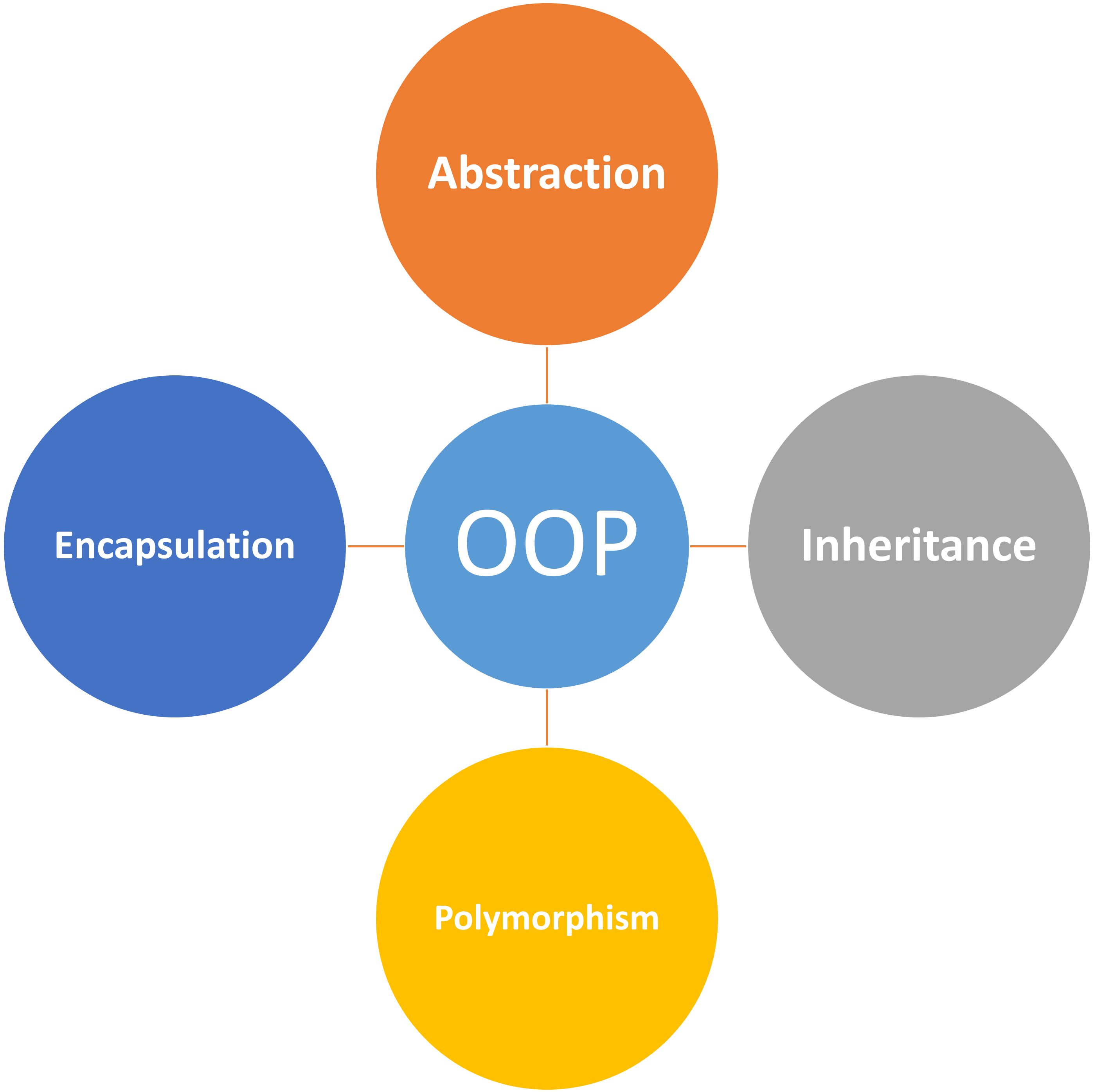
And Llama

And Perplexity

Objectives

How to :

- Create classes to define objects
- Write methods and create attributes for objects
- Instantiate objects from classes and initialize them
- Restrict access to object attributes
- Use inheritance and polymorphism



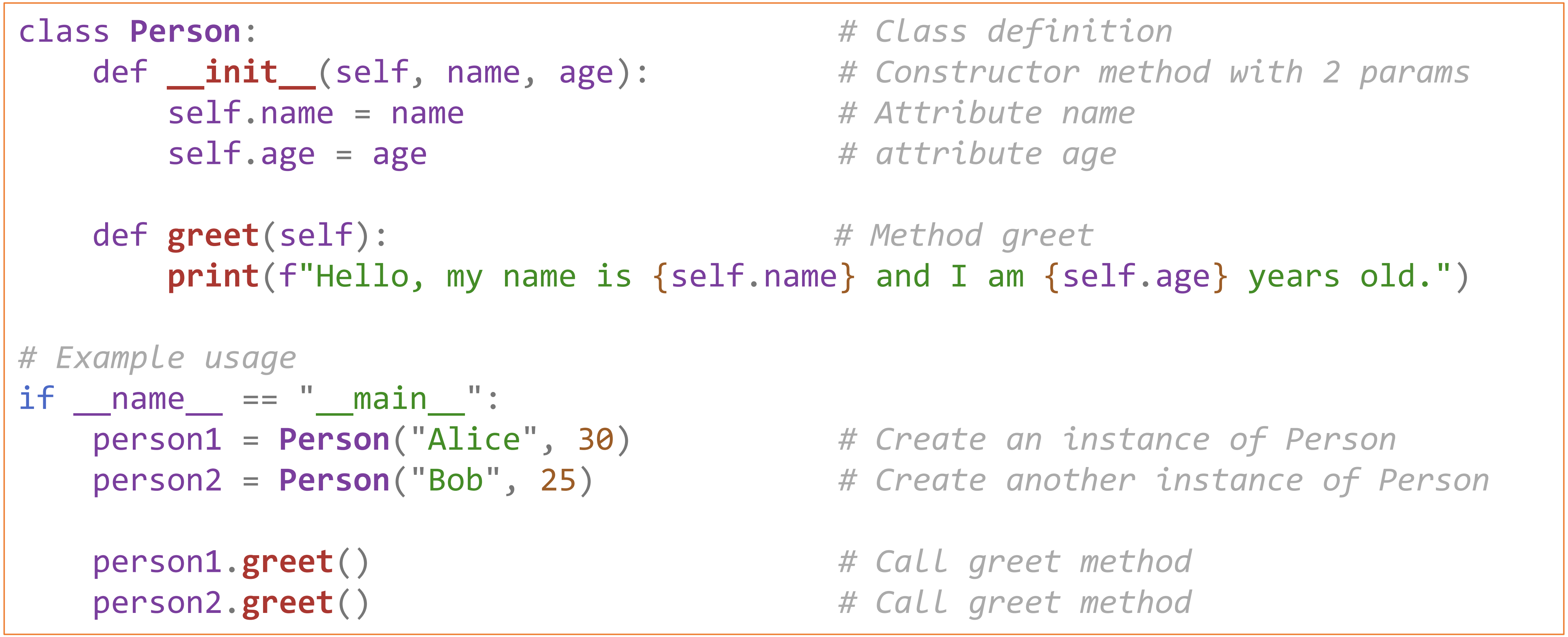
Abstraction

Encapsulation

OOP

Inheritance

Polymorphism



```python
class Person:                               # Class definition
    def __init__(self, name, age):          # Constructor method with 2 params
        self.name = name                    # Attribute name
        self.age = age                      # attribute age

    def greet(self):                        # Method greet
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Example usage
if __name__ == "__main__":
    person1 = Person("Alice", 30)           # Create an instance of Person
    person2 = Person("Bob", 25)             # Create another instance of Person

    person1.greet()                         # Call greet method
    person2.greet()                         # Call greet method
```

Key concepts

- **Class:** A template of a concept with its properties (attributes) and behaviour (methods).
- **Object:** An instance of a class. Concrete entity created from the class.

- **Encapsulation:** Bundling data and methods (code) within a single entity (class).
- **Abstraction:** Hide details and show only essential information.
- **Inheritance:** A class can inherit attributes and methods from another class.
- **Polymorphism:** The ability to use the same method name with different behavior.
- **Composition:** An attribute can be an object of another class

Constructor : __init__

- The method __init__ initializes the class object.
- It is automatically called every time the class is instantiated
- It can have parameters that allow you to initialize different attributes

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


cat = Animal('Meow')
print(cat.voice)      # Output: Meow

dog = Animal('Woof')
print(dog.voice)      # Output: Woof
```

The convention self

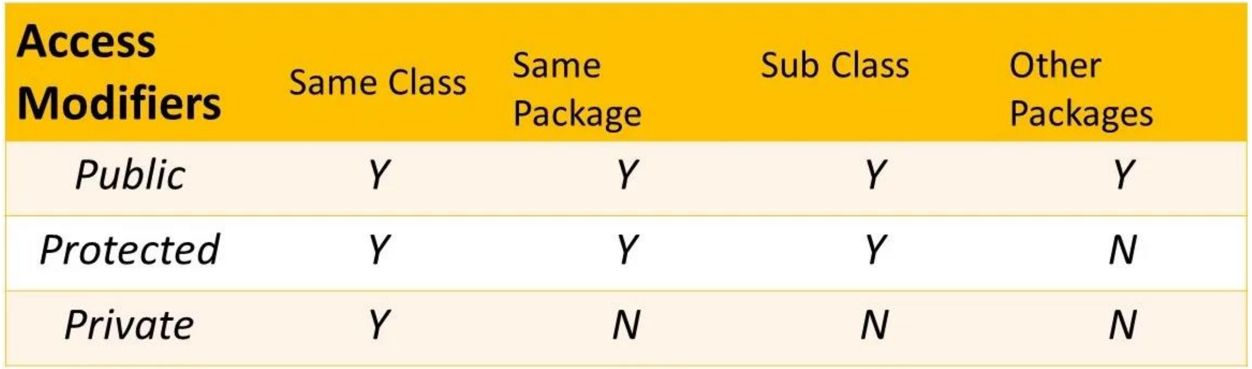
Class methods have only one specific difference from ordinary functions

- they have an extra variable that has to be added to the beginning of the parameter list
- but we do not give a value for this parameter when we call the method.
- this particular variable refers to the object itself,
- and by convention, it is given the name **self**.

```
class Counter:
    def __init__(self, start=0):
        self.count = start

    def increment(self):
        self.count += 1

    def decrement(self):
        self.count -= 1

    def reset(self):
        self.count = 0

    def get_count(self):
        return self.count

    def set_count(self, count):
        self.count = count
```

Access Modifiers : Public, private and protected

- All member variables and methods are public by default in Python
- Protected : By prefixing the name of your member with **a single underscore**
- Private : prefixed with at least **two underscores**



Access Modifiers	Same Class	Same Package	Sub Class	Other Packages
Public	*Y*	*Y*	*Y*	*Y*
Protected	*Y*	*Y*	*Y*	*N*
Private	*Y*	*N*	*N*	*N*

```
class Test:
    varPublic = 10
    _varProtected = 20
    __varPrivate = 30

    def publicMethod(self):
      print("Public Method")

    def _protectedMethod(self):
      print("Protected Method")

    def __privateMethod(self):
      print("Private Method")
```

Inheritance

- Inheritance mechanism allows you to create a new class from an existing class.
    - Child class = Subclass
    - Parent class = Superclass
- Child can add attributes and methods
- It can also rewrite the methods of the parent class → Method overriding

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

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Bird(Animal):
    def speak(self, repeat=1):
        return "Tweet! " * repeat

if __name__ == "__main__":
    dog = Dog("Rex")
    print(dog.speak())

    bird = Bird("Tweety")
    print(bird.speak(3))
```

Polymorphism

- Manipulating elements that share the same parents
- The same method name can have different behaviors based on the class that is being used
- Polymorphism allows to call the suitable methods depending on the object

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

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Bird(Animal):
    def speak(self, repeat=2):
        return "Tweet!" * repeat

if __name__ == "__main__":
    animals = [Dog("Rex"), Bird("Tweety")]
    for animal in animals:
        print(animal.speak())
```

Overloading operators

```python
class Vector:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y, self.z + other.z)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar, self.z * scalar)

    def __repr__(self) -> str:
        return f'Vector({self.x}, {self.y}, {self.z})'

# Example usage:
v1 = Vector(1, 2, 3)
v2 = Vector(4, 5, 6)

print(v1 + v2)  # Vector(5, 7, 9)
print(v1 * 3)   # Vector(3, 6, 9)
```

Overloading arithmetic operators

Arithmetic operator (Operation)	**Magic method**
+ (Addition)	__add__(self, other)
- (Subtraction)	__sub__(self, other)
* (Multiplication)	__mul__(self, other)
/ (Division)	__truediv__(self, other)
% (Modulo)	__mod__(self, other)
** (Power)	__pow__(self, other)

Overloading comparison operators

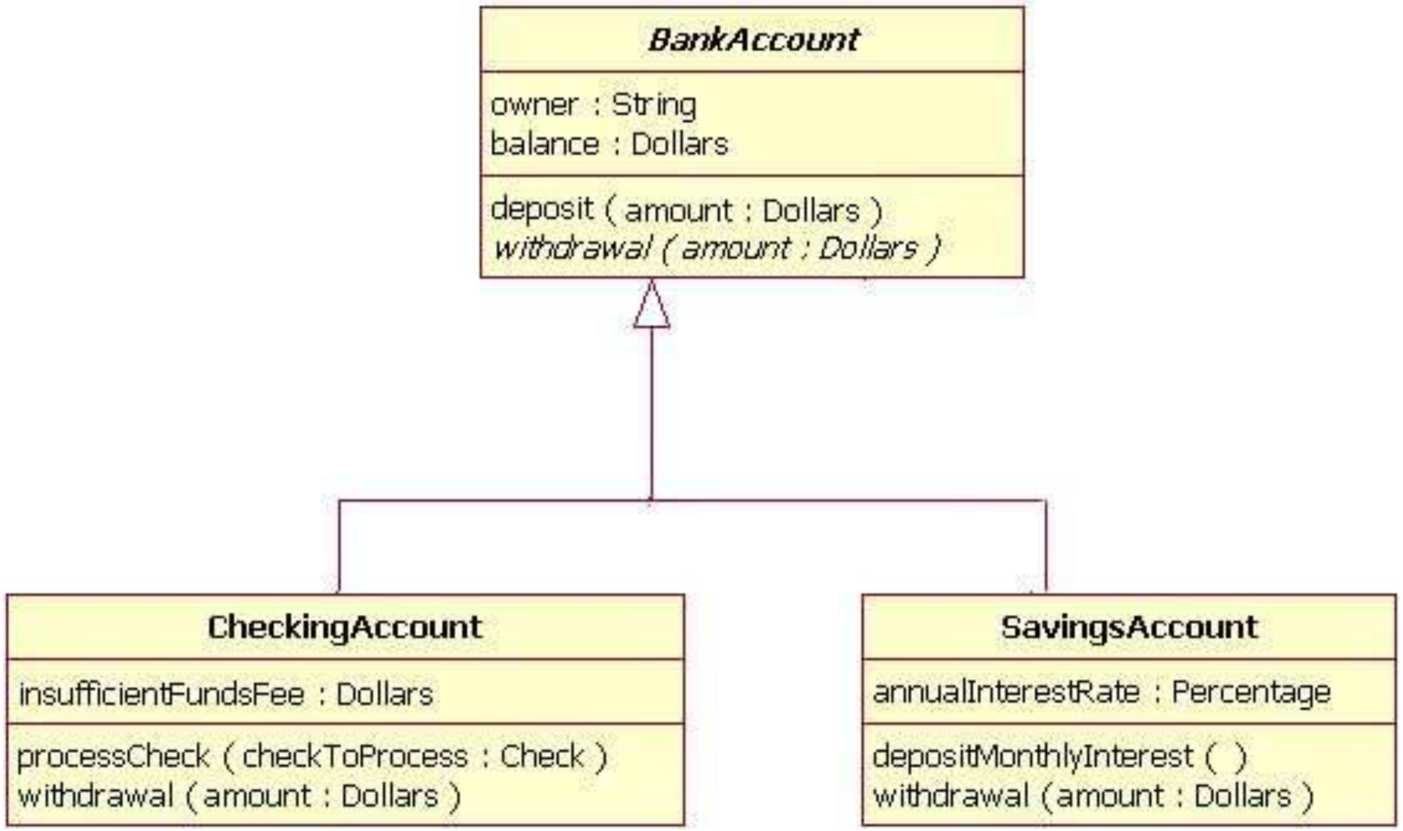
Comparison operator (Operation)	Magic method
< (Less than)	__lt__(self, other)
> (Greater than)	__gt__(self, other)
<= (Less than or equal to)	__le__(self, other)
>= (Greater than or equal to)	__ge__(self, other)
== (Equal)	__eq__(self, other)
!= (Not equal)	__ne__(self, other)

Augmented Assignments

Operator	**Magic method**
+=	.__iadd__(self, other)
-=	.__isub__(self, other)
*=	.__imul__(self, other)
/=	.__itruediv__(self, other)
//=	.__ifloordiv__(self, other)
%=	.__imod__(self, other)
**=	.__ipow__(self, other[, modulo])

Exercise 1

- Implement with examples and history of transactions



BankAccount

owner : String
balance : Dollars

deposit (amount : Dollars)
withdrawal (amount : Dollars)

CheckingAccount

insufficientFundsFee : Dollars

processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)

SavingsAccount

annualInterestRate : Percentage

depositMonthlyInterest ( )
withdrawal (amount : Dollars)

Exercise 2

- Implement different shape classes (rectangle, circle, square, …)
- Add a method computeArea() that computes the area of each one of them
- Create a list of 100 random shapes.
  Display their represantation and compute the sum of areas.

Exercise 3

- Implement the class Fractional and its operators

```
# Example usage:
# frac1 = Fractional(1, 2)
# frac2 = Fractional(3, 4)
# print(frac1 + frac2)  # Output: 5/4
# print(frac1 > frac2)  # Output: False
# fract3 = Fractional(42, 0)   ???
```

Now, you can reactivate Copilot!