😊 Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into reusable **objects**. An object is an instance of a **class**, which is like a blueprint defining attributes (data) and methods (functions).

Key Concepts of OOP

1. **Classes and Objects**
    - A **class** is a template for creating objects.
    - An **object** is an instance of a class with its own unique data.
2. **Encapsulation**
    - Bundles data and methods within a class to restrict direct access.
    - Protects object integrity by controlling modifications.
3. **Inheritance**
    - Allows a class to inherit attributes and methods from another class.
    - Promotes code reusability and hierarchy.
4. **Polymorphism**
    - Enables different classes to have methods with the same name but different behaviors.
    - Improves flexibility and scalability.
5. **Abstraction**
    - Hides complex implementation details and exposes only essential features.
    - Enhances code simplicity and security.

Use Cases of OOP

1. **Software Development** – Used in web applications (Django, Flask), game development (Pygame), and desktop applications (Tkinter).
2. **Data Science and Machine Learning** – Frameworks like TensorFlow and Scikit-learn are built using OOP.
3. **Database Management** – ORM (Object-Relational Mapping) tools like SQLAlchemy use OOP principles.
4. **Automation and Scripting** – Helps create modular and reusable automation scripts.
5. **Enterprise Applications** – Large-scale applications use OOP for better organization and maintainability.

Benefits of OOP

- **Code Reusability** – Inheritance allows reusing existing code without rewriting.
- **Scalability** – Easy to extend applications with new features.
- **Maintainability** – Modular structure simplifies debugging and updating.
- **Data Security** – Encapsulation prevents accidental modifications.
- **Code Organization** – Objects group related data and functions, improving readability.

😄 To begin with the Lab

1. First, let's start with creating a list and then getting the count for number 2, that how many times it has occurred in the list.

```
[1]: lst = [1,2,3]

     Remember how we could call methods on a list?

[2]: lst.count(2)

[2]: 1
```

2. In Python, ***everything is an object***. Remember from previous labs, we can use type () to check the type of object something is.

```
[3]: print(type(1))
     print(type([]))
     print(type(()))
     print(type({}))

     <class 'int'>
     <class 'list'>
     <class 'tuple'>
     <class 'dict'>
```

3. So, we know all these things are objects, so how can we create our own Object types? That is where the **class** keyword comes in.
4. User defined objects are created using the class keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class.
5. This code creates a blueprint called "Sample" and then makes an object "x" using that blueprint. Finally, it prints the type of "x" to confirm that it is an instance of the "Sample" class.

[4]:
```
# Create a new object type called Sample
class Sample:
    pass

# Instance of Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

6. The below code defines a new object type called Dog. When you create a Dog, the initialization method runs, setting the dog's breed based on the value provided. Two Dog objects are then created: one with the breed set to "Lab" and another with the breed set to "Huskie".

[5]:
```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

[6]:
```
sam.breed
```

[6]:
```
'Lab'
```

[7]:
```
frank.breed
```

[7]:
```
'Huskie'
```

7. In the code below we are saying that a Dog class with a **class-level attribute** species set to 'mammal'. Every Dog object has a **breed** and a **name**, which are assigned during initialization. When you create sam = Dog('Lab','Sam'), it has the **breed** 'Lab' and **name** 'Sam', and it also inherits the **species** 'mammal' from the class.

[8]:
```
class Dog:

    # Class Object Attribute
    species = 'mammal'

    def __init__(self,breed,name):
        self.breed = breed
        self.name = name
```

[9]:
```
sam = Dog('Lab','Sam')
```

[10]:
```
sam.name
```

[10]:
```
'Sam'
```

[11]:
```
sam.species
```

[11]:
```
'mammal'
```

8. **Methods** are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. **Methods** are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.
9. The code below defines a blueprint for a circle. It sets a constant value for pi and allows you to create a circle with a certain radius. When a circle is created, its area is calculated. There are methods to change the radius (which recalculates the area) and to compute the circumference.

[12]:
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

10. Now let's change the radius and see how that affects our Circle object: After calling c.setRadius(2), the circle's radius changes from 1 to 2. This updates the area to 2^2 * 3.14 (12.56) and the circumference to 2 * 2 * 3.14 (12.56). The new values are then printed.

[13]:
```
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

11. **Inheritance** is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes; the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of the complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).
12. Using the code below we are demonstrating **inheritance** in object-oriented programming. The base class "Animal" has methods to announce creation, identify itself, and simulate eating. The "Dog" class inherits from Animal, calls the Animal constructor to ensure proper setup, then adds its own behaviour by printing "Dog

created," overriding the identification method to say "Dog," and adding a new "bark" method.

[14]:
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

[15]:
```
d = Dog()
```

```
Animal created
Dog created
```

[16]:
```
d.whoAmI()
```

```
Dog
```

[17]:
```
d.eat()
```

```
Eating
```

[18]:
```
d.bark()
```

```
Woof!
```

13. We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, ***polymorphism*** **refers** to the way in which different object classes can share the same method name, and those methods can be called from the same place, even though a variety of different objects might be passed in.
14. This code defines two separate classes, one for a dog and one for a cat. Both classes have a method called **speak ()** that returns a unique message for each animal using their name. When you create a Dog named "Niko" and a Cat named "Felix," calling their **speak ()** methods prints their respective messages.

[19]:
```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

15. The code demonstrates polymorphism. It loops through a list of pet objects (a dog and a cat) and prints the result of each object's speak () method, which behaves according to the object's type. It also defines a function that takes any pet object and prints its speak message, achieving the same result.

[20]:
```
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Another is with functions:

[21]:
```
def pet_speak(pet):
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

16. Now we are defining an abstract base class "Animal" with a constructor that sets a name and a "speak" method that is meant to be overridden by subclasses. If a subclass does

not implement "speak", calling it will raise a NotImplementedError. The "Dog" and "Cat" classes inherit from Animal and provide their own implementations of "speak". When you create a Dog named "Fido" and a Cat named "Isis", calling their "speak" methods returns "Fido says Woof!" and "Isis says Meow!", respectively.

[22]:
```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

17. So, now in the end, we are going over some special methods in Python. Classes in Python can implement certain operations with special method names. These methods are not actually called directly, but by Python-specific language syntax.
18. For example, let's create a class called Book. To better understand, this code defines a Book class that uses special methods to customize its behaviour. The constructor initializes a book with a title, author, and page count.
19. A string method returns a formatted description when the book is printed, a length method returns its page count, and a destructor prints a message when the book is deleted.

[23]:
```python
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```

[24]:
```python
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```