ILP 2024 : Computing

W3S1
27 May 2024

Class and Objects

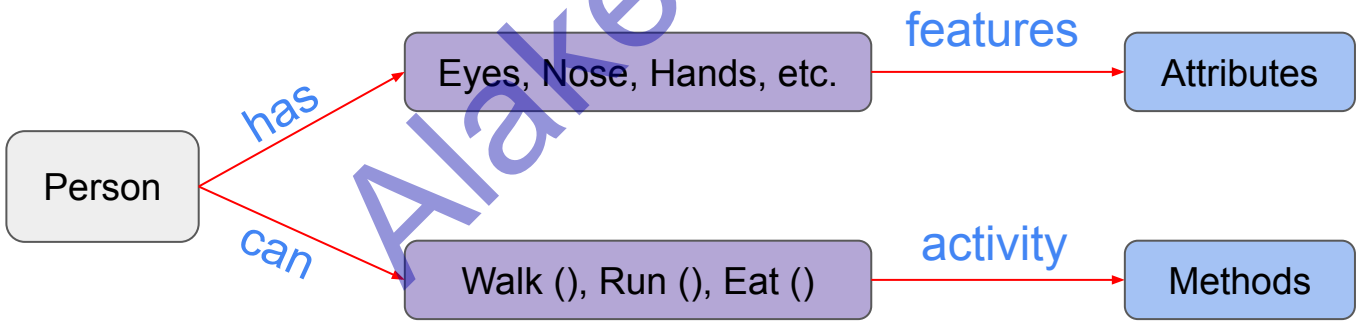
Objectives

- OOP concept
- Class and Object
- Attribute, method, special methods etc.
- Inheritance
- Polymorphism
- Encapsulation

Introduction to OOP

- Python is an **object-oriented programming** (OOP) language, meaning it allows you to model real-world entities using **classes** and **objects**.

- Classes are **blueprints** for creating objects. They define the properties (attributes) and behaviors (methods) of objects.



- Sometimes, the objects (person/car) we need for our programs cannot be described using only the basic int/float/str/list/dict types.

Contd…

- A class named "Car" could represent the blueprint for all cars make by a company.
- Attributes of the Car class could include properties like `model`, `color`, `mileage`, and many more
- Methods of the Car class could include functions like `start()`, `stop()`, `accelerate()`, and `brake()`.
- **Each individual car** made by the company would be an **object** instantiated from the **Car** class, with its specific make, model, color, etc.
- Just as in real life, where cars have unique characteristics and can perform actions, in Python, each **car object** has its own set of **attributes** and **can execute methods** defined within the Car class.
- *We often prefer to create our own custom types/objects using the concept of OOP*.

Contd…

- Bank “Account” is a class
- Account number, account holder `name`, `balance`, and `account type` are attributes
- `deposit()`, `withdraw()`, `check_balance()`, and `transfer()` are methods
- Each individual account held by a customer would be an object instantiated from the Account class,

- “Student” is a class
- Attributes:
  `Student_id`, `name`, `age`
  `attendance_record`: Dictionary, where keys are dates and values are attendance status ('Present', 'Absent', etc.).
- `get_student_id()`, `get_name()`, `get_age()`, `get_grade()`

Syntax to create a Class

```
class ClassName:

    ...

        Define Attributes

        Define Methods

    ...
```

```
class Car:
    Make = "BMW"
    Model = "C2500"
    speed = 0 #in kmph

    def start():
        print("Starting the car")
        Car.speed = 5

    def stop():
        print("Stopping the car")
        Car.speed = 1

my_car = Car

print(type(my_car))
print(my_car)

print("----------------------")
print("Speed of my car is: ", my_car.speed)

print("----------------------")
my_car.start()
print("Speed of my car is: ", my_car.speed)

print("----------------------")
my_car.stop()
print("Speed of my car is: ", my_car.speed)
```

```
<class 'type'>
<class '__main__.Car'>
----------------------
Speed of my car is:  0
----------------------
Starting the car
Speed of my car is:  5
----------------------
Stopping the car
Speed of my car is:  1
```

Defining a Class

Defining Attributes

Defining Methods

Create an Object

Accessing an attribute

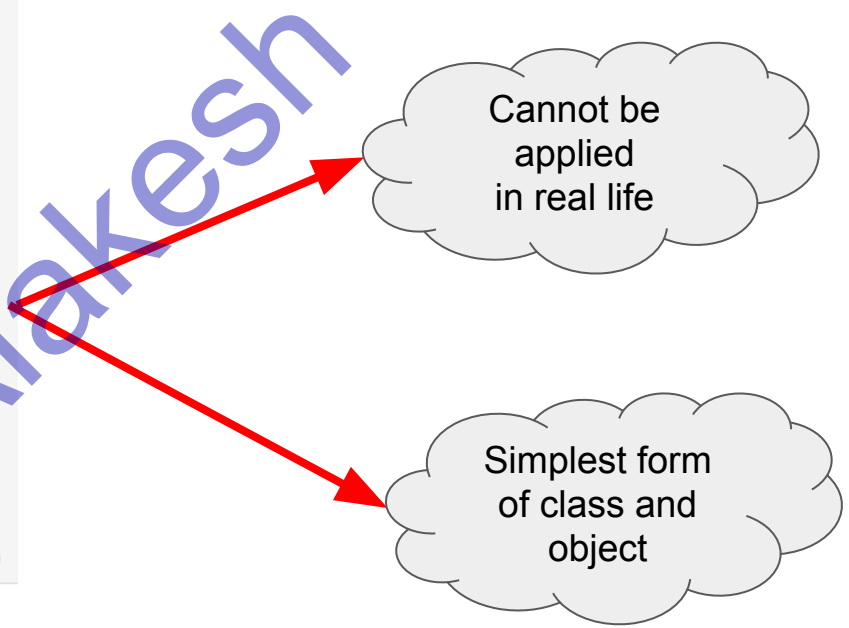
Called a method

How to create a new car, say `my_second_car`

Contd …

```
class Car:
    Make = "BMW"
    Model = "C2500"
    speed = 0 #in kmph

    def start():
        print("Starting the car")
        Car.speed = 5

    def stop():
        print("Stopping the car")
        Car.speed = 1

my_car = Car
my_second_car = Car


my_car.start()
print("Speed of my car is: ", my_car.speed)

print("----------------------")
print("Speed of my second car is: ", my_second_car.speed)
```

```
Starting the car
Speed of my car is:  5
----------------------
Speed of my second car is:  5
```



The `__init__()` Function

```python
#intoduction to __init__() function

class Car:
    def __init__(self, make, model, speed):
        self.make = make
        self.model = model
        self.speed = speed


    def start(self):
        print("Starting the car")
        self.speed = 5

    def stop(self):
        print("Stopping the car")
        self.speed = 1

my_car = Car("BMW", "C2500", 0)
my_second_car = Car("Audi", "A5 Sportback", 0)


my_car.start()
print(f"I have a {my_car.make}: {my_car.model} car, current speed is: {my_car.speed}")

print("----------------------")
print(f"I have another {my_second_car.make}: {my_second_car.model} car, current speed is: {my_second_car.speed}")
```

- Initialize the newly created object
- Automatically invoke

- `self` is a reference to the current instance of the class
- `self` is used only inside the class definition


```
Starting the car
I have a BMW: C2500 car, current speed is: 5
----------------------
I have another Audi: A5 Sportback car, current speed is: 0
```

The "magic" methods: `__init__()`

- The double underscores make `init` a special method.
- **Clarity and Readability:** the use of double underscores enhances the clarity and readability of the code. It clearly indicates to other developers that this method is special.
- These special methods are **invoked implicitly** by Python in response to specific operations or built-in functions.
- Using double underscores helps avoid name clashes with user-defined methods or attributes.

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.is_running = False
        self.speed = 0

    def start(self):
        if not self.is_running:
            print("Starting the car.")
            self.is_running = True
        else:
            print("The car is already running.")

    def stop(self):
        if self.is_running:
            print("Stopping the car.")
            self.is_running = False
            self.speed = 0
        else:
            print("The car is already stopped.")

    def accelerate(self, speed_increase):
        if self.is_running:
            self.speed += speed_increase
            print(f"Accelerating. Current speed: {self.speed} mph.")
        else:
            print("Cannot accelerate, the car is not running.")

    def brake(self, speed_decrease):
        if self.is_running:
            if self.speed - speed_decrease >= 0:
                self.speed -= speed_decrease
                print(f"Applying brakes. Current speed: {self.speed} mph.")
            else:
                print("Speed cannot be negative. Applying full brakes.")
                self.speed = 0
        else:
            print("Cannot apply brakes, the car is not running.")

# Creating an instance of the Car class
my_car = Car("Toyota", "Camry", 2022)

# Starting the car
my_car.start()
my_car.start()

# Accelerating
my_car.accelerate(30)

# Applying brakes
my_car.brake(40)

# Stopping the car
my_car.stop()
```


output

```
Starting the car.
The car is already running.
Accelerating. Current speed: 30 mph.
Speed cannot be negative. Applying full brakes.
Stopping the car.
```

```python
print(my_car.__dict__)
```

```
{'make': 'Toyota', 'model': 'Camry', 'year': 2022, 'is_running': False, 'speed': 0}
```

Let’s design a game :)

Enemy

Army

```
Round 13:
Army wins this round!
Army's points: 50, Enemy's points: 70

Round 14:
Enemy wins this round!
Army's points: 50, Enemy's points: 80
```

- We have a **Player** class representing both the **army** and the **Enemy**.
- Each player has attributes for **power** and **points**.
- The game is simulated within a loop where players **generate random power** between 1 and 10.
- The player with the **higher power in each round** earns 1**0 points**.
- The game continues until one player's **power** becomes **zero** or **after 100 rounds**.
- Finally, the **winner** is determined based on the player with the **higher power**, and their points are printed.

Practise test

Can you write a program to build a robot using robot class. The robot should be in the initial/current coordinates (0, 0). Now ask user to enter (L, R, U, D) to move the robot one step towards left, right, up and down, respectively. For example, if the user press “R”, then the robot moves to (0,1) position. Coordinates of the robot should always be in between (0,0) and (4,4).


```
Current Position: (0, 0)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  L
Cannot move left. Already at the leftmost position.
Current Position: (0, 0)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  R
Current Position: (0, 1)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  R
Current Position: (0, 2)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  R
Current Position: (0, 3)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  R
Current Position: (0, 4)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  R
Cannot move right. Already at the rightmost position.
Current Position: (0, 4)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  U
Cannot move up. Already at the topmost position.
Current Position: (0, 4)
Enter 'L' for left, 'R' for right, 'U' for up, 'D' for down, or 'Q' to quit:  Q
Exiting the program.
```

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

    def move_left(self):
        if self.y > 0:
            self.y -= 1
        else:
            print("Cannot move left. Already at the leftmost position.")
```

The `__str__()` Function

The `__str__()` function controls what should be returned when the class object is represented as a string.

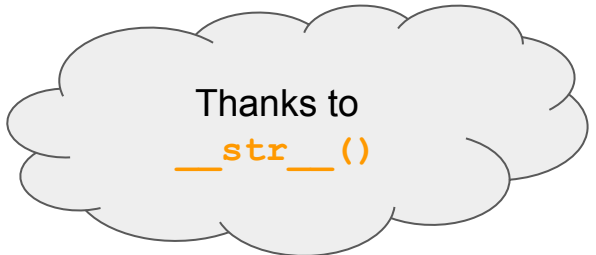
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1)
```

```
<__main__.Person object at 0x103ba1a00>
```



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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

```
John(36)
```



The `__add__()`

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

    def __add__(self, other):
        if isinstance(other, Coordinate):
            new_x = self.x + other.x
            new_y = self.y + other.y
            return Coordinate(new_x, new_y)
        else:
            print("Both are not from same class")

    def __str__(self):
        return "({}, {})".format(self.x, self.y)

# Creating instances of Coordinate
coord1 = Coordinate(3, 4)
coord2 = Coordinate(1, 2)

# Adding coordinates using the __add__ method
result = coord1 + coord2

# Displaying the result
print("Result of addition:", result)
```

```
Result of addition: (4, 6)
```


`__sub__()`: -
`__mul__()`: *
`__truediv__()`: /
`__floordiv__()`: //
`__mod__()`: %
`__pow__()`: **
`__eq__()`: ==
`__ne__()`: !=
`__lt__()`: <
`__le__()`: <=
`__gt__()`: >
`__ge__()`: >=

`__str__()`: Invoked when the `str()` function is called on an object to get its string representation.

`__len__()`: Invoked when the `len()` function is called on an object to get its length.

The `pass` statement

Class definitions **cannot be empty**, but if you for some reason have a class definition with no content, put in the `pass` statement to avoid getting an error.

```
class Person:
    pass
```

To **delete an objects**, we need to use the `del` keyword:

```
del P1
```

Inheritance

- Our robot can move L, R, U and D. But now, we want another robot who can also clean the floor.
- We can **inherit** the existing attributes and methods of our previous robot to our new cleaning robot.
- **Inheritance** allows us to define a class that **inherits** all the methods and properties **from another class**.
    - **Parent class** is the class being inherited from, also called **base class**.
    - **Child class** is the class that inherits from another class, also called **derived class**.


```
class Child_Class(Parent_Class):
    ...
```

Contd …

```python
#inheritance
#Subclass inheriting from Robot

class CleaningRobot(Robot):
    def clean(self):
        print("Cleaning the floor.")

# Create an instance of CleaningRobot
cleaning_robot = CleaningRobot()

# Set coordinates using the superclass method
cleaning_robot.set_coordinates(2, 3)

# Access coordinates using the superclass method
print("Current Position (using getter):", cleaning_robot.get_coordinates())

# Move the robot using the superclass methods
cleaning_robot.move_left()
cleaning_robot.move_up()

# Call the method specific to CleaningRobot
cleaning_robot.clean()
```

Need one more attribute, say `active`


```
Current Position (using getter): (2, 3)
Cleaning the floor.
```

Contd …

```python
class CleaningRobot(Robot):
    def __init__(self):
        super().__init__()  # Call superclass constructor
        self.active = False  # Additional attribute for cleaning status

    def clean(self):
        if not self.active:
            print("Starting cleaning.")
            self.active = True
        else:
            print("Already cleaning.")

    def stop_cleaning(self):
        if self.active:
            print("Stopping cleaning.")
            self.active = False
        else:
            print("Not cleaning. Already stopped.")
```

`super()` inherits from the parent


```
Current Position (using getter): (2, 3)
Starting cleaning.
Stopping cleaning.
```

Polymorphism

The word "**polymorphism**" means "**many forms**", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Method overriding: It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

Operator overriding: It allows operators to have different implementations depending on the operands.


```
#Polymorphism

class Animal:
    def sound(self):
        print("Animal makes a sound")
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

    def __str__(self):
        return f"Value of X: {self.x} and value of Y: {self.y}"

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3)
print("-----------------")
print(5+6)
```

```
Value of X: 4 and value of Y: 6
-----------------
11
```

```
Dog barks
Cat meows
```

Attribute types

Python allows control over **access to** class **attributes** through the concepts of *private*, *public*, and *protected* attributes.

Public attributes:

- These are accessible from outside the class.
- They can be accessed, modified, and deleted directly.
- By default, attributes are public


```python
#Public

class BankAccount:
    def __init__(self, account_number):
        self.account_number = account_number
        self.balance = 0  # Public attribute

account = BankAccount("123456789")
print(account.balance)
```

```
0
```

Contd …

Private attributes:

- These are not accessible from outside the class.
- They are denoted by prefixing the attribute name with double underscores (__).

```
class BankAccount:
    def __init__(self, account_number):
        self.__account_number = account_number
        self.__balance = 0  # Private attribute

account = BankAccount("123456789")
print(account.__balance)  # Raises AttributeError
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent c
Cell In[9], line 7
      4         self.__balance = 0  # Private attribute
      6 account = BankAccount("123456789")
----> 7 print(account.__balance)  # Raises AttributeError

AttributeError: 'BankAccount' object has no attribute '__balance'
```



```
class BankAccount:
    def __init__(self, account_number):
        self.__account_number = account_number
        self.__balance = 0  # Private attribute

    def get_balance(self):
        return self.__balance

    def set_balance(self, new_balance):
        if new_balance >= 0:
            self.__balance = new_balance

account = BankAccount("123456789")
account.set_balance(1000)     # Modifying private attribute using setter method
print(account.get_balance())  # Accessing private attribute using getter method
```

getter

setter

```
1000
```

Define getter and setter methods within the class to access and modify private attributes, respectively.

Contd …

Protected attributes

- These are accessible from within the class and its subclasses.
- They are denoted by prefixing the attribute name with a single underscore (_).

```
class BankAccount:
    def __init__(self, account_number):
        self._account_number = account_number
        self._balance = 450  # Protected attribute

class SavingsAccount(BankAccount):
    def __init__(self, account_number):
        super().__init__(account_number)
        self.interest_rate = 0.05  # Public attribute

savings_account = SavingsAccount("987654321")
print(savings_account._balance)  # Output: 0
```

```
class BankAccount:
    def __init__(self, account_number):
        self._account_number = account_number
        self.__balance = 450  # Protected attribute

class SavingsAccount(BankAccount):
    def __init__(self, account_number):
        super().__init__(account_number)
        self.interest_rate = 0.05  # Public attribute

savings_account = SavingsAccount("987654321")
print(savings_account._balance)  # Output: 0
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[16], line 12
      9         self.interest_rate = 0.05  # Public attribute
```

Private attributes cannot be accessed outside class

Note: Accessing protected attributes directly from outside the class or its subclasses is discouraged, although technically possible.

Encapsulation

- It involves **bundling** the data (attributes) and methods (functions) that operate on the data into a single unit, known as a class.
- Encapsulation allows for data hiding, abstraction, promoting better code organization and modularity.
- In Python, encapsulation is achieved through the use of classes and access modifiers.

The `property` keyword

```python
class BankAccount:
    def __init__(self, account_number):
        self._account_number = account_number
        self.__balance = 450  # Private attribute

    def get_balance(self):
            return self.__balance

class SavingsAccount(BankAccount):
    def __init__(self, account_number):
        super().__init__(account_number)
        self.interest_rate = 0.05  # Public attribute

savings_account = SavingsAccount("987654321")
print(savings_account.get_balance())  # Output: 0
```

```
450
```

We need to use getter or setter to access/modify private attributes

The `property` provides a **flexible** way to access private attributes within a class

```python
class BankAccount:
    def __init__(self, account_number):
        self._account_number = account_number
        self.__balance = 450  # Private attribute

    def get_balance(self):
            return self.__balance

    balance = property(get_balance)

class SavingsAccount(BankAccount):
    def __init__(self, account_number):
        super().__init__(account_number)
        self.interest_rate = 0.05  # Public attribute

savings_account = SavingsAccount("987654321")

print(savings_account.get_balance())  # Output: 0
print(savings_account.balance)
```

```
450
450
```

Thanks to Property !!!

Conclusion