

## 01. OBJECT ORIENTED PROGRAMMING

- It's a programming paradigm
- In OOP we have classes and objects

**Encapsulation**  
Putting things together

**Abstraction**  
Hiding the irrelevant features

### 4 PILLARS OF OOP

**Inheritance**  
objects of one class can inherit the features of another

**Polymorphism**  
Same entity but different behavior

## CLASS

- Class is a **blueprint** that defines the **methods and properties** of an object.
- self** argument is the reference to the object itself

```
class Dog:
    kind = "canine"
    def __init__(self, name):
        self.name = name
```

- Private properties:** only accessible inside class definition.
- append '\_' as a prefix in property name to make it **private**.

**Note:** Nothing can be completely private in python.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance
```

## CONSTRUCTOR

- It's the **first function** that'll be called whenever an object is created.
- Python doesn't provide direct access to constructors

**Note:** `__init__` is not the constructor, it's initializing function

## DUNDER/MAGIC METHODS

- methods in class can be modified/overloaded to change the default behavior of that object.
- some useful magic methods

- modifying print() behavior

`__str__(self)`

- modifying call behavior

`__call__(self)`

- addition behavior

`__add__(self, other)`

- less than operator behavior

`__lt__(self, other)`

## 02. INHERITANCE

- Objects of one class can inherit the features of another

```
class Parent:
    def __init__(self):
        print("parent class")

class child(Parent):
    def __init__(self):
        print("child class")

# inheriting from Parent class
```

- super()** method is used to call methods of parent class

```
class child(Parent):
    def __init__(self):
        super().__init__()
        print("child class")
```

```
class A:
    def __init__(self, a):
        self.a = a

class B:
    def __init__(self, b):
        self.b = b

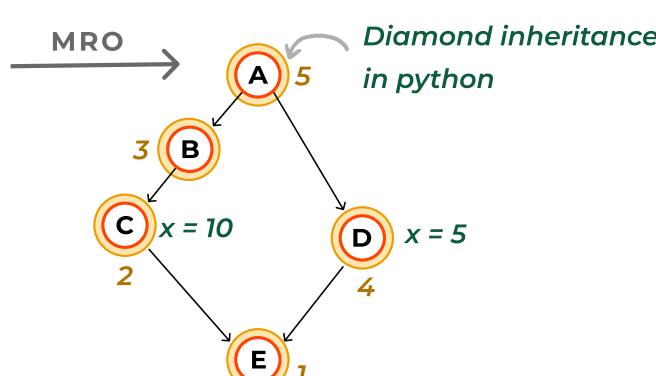
# C inherits from both A and B
class C(A, B):
    def __init__(self, a, b, c):
        A.__init__(self, a)
        B.__init__(self, b)
        self.c = c
```

## 03. METHOD RESOLUTION ORDER

- It is the order in which a method is searched for in a classes hierarchy

### Rules:

- Left to right
- visit parent when all its children are already visited



## METHOD RESOLUTION ORDER

- left to right
- We go to a parent when all its children are considered

**e = E()**

**E** → **C** → **B** → **D** → **A**

## 04. LAMBDA FUNCTIONS

Short syntax to write functions in Python

```
lambda arg1, arg2 :
    return "output"
```

## 05. HIGHER ORDER FUNCTIONS

A function that returns function

```
def outer_fn(n):
    def inner_fn(x):
        return x+2
    return inner_fn
```

## 06. DECORATORS

High order functions that take another function as input and add the extra behavior in along with the functionality of passed function

```
def pretty(func):
    def inner():
        print("-" * 50)
        func()
        print("-" * 50)
    return inner

@pretty # usage
def foo():
    print("WHATTTT?")
```

## 07. PRINCIPLES OF FUNCTIONAL PROGRAMMING

- Data should be separated from mutations
- Treat variable as immutable
- Treat functions as FCC

## MAPS

Takes multiple iterables and perform some function on them and returns output map

```
map(function_to_perform, *iterables)
# example:
list(map(lambda x: x**2, [1,2,3]))
Output : [1,4,9]
```

## FILTER

Filter out elements from a list on the basis of some condition

```
list(filter(lambda x: x%2 == 0,
[1,2,3,4,5]))
Output : [2,4]
```

## ZIP

Returns an iterator that will aggregate elements from two or more iterables

```
a = [1,2,3]
b = ["a", "b", "c", "d", "e"]
list(zip(b,a))
Output:[('a', 1), ('b', 2), ('c', 3)]
```

## REDUCE

Reduces an iterable into single value

```
from functools import reduce
reduce(lambda x, y: x + y, [1,2,3])
Output : 6
```

## ARGS AND KWARGS

**args** are variable size positional arguments stored inside tuple

```
def sum_number(x, y, *args):
    result = x + y
    if args:
        result += sum(args)
    return result
```

**kwargs** are variable size keyword arguments stored inside dictionary

```
def create_person(name, age, gender,
                  **extra_info):
    Person = {
        "name": name,
        "age": age,
        "gender": gender
    }

    if extra_info:
        Person.update(extra_info)
    return Person
```

Order of passing arguments: Positional -> Args -> Keyworded -> Kwargs

## 08. FILE HANDLING

- Use secondary memory, to keep data even when program terminated
- Everything in memory is a sequence of bytes or byte array.

## FILE ACCESS MODES

Read only <b>r</b>	Read and Write <b>r+</b>	Read binary <b>rb</b>	Append and Read <b>a+</b>
Write only <b>w</b>	Write and Read <b>w+</b>	Write binary <b>wb</b>	Append Only <b>a</b>

## WORKING WITH FILES

Opening a file	<code>file = open("sample.txt", "r")</code>
Closing a file	<code>file.close()</code>
Writing to a file	<code>file = open("sample.txt", "w+") file.writelines(["1\n", "2\n"]) # write from list file.write("hellow world") # write from string file.close()</code>
Reading from a file	<code>file = open("sample.txt", "r+") file.read() # read everything as string file.readline() # read line by line file.readlines() # read all lines as list file.close()</code>
Reading large files	<code>file = open("sample2.txt", "r+") buffer = file.readline() while buffer:     # n lines = 1 block of memory     print(buffer, end = "")     buffer = file.readline() file.close()</code>
Moving reading/ writing cursor	<code>file.seek(3) # moving 3 characters ahead</code>
Smart way of working with files.  “with” statement simplifies exception handling by encapsulating common preparation and cleanup tasks.”	<code>with open("sample3.txt", "r+") as file: print(file.read(5)) file.seek(0) print(file.read())</code>

## 09. MODULES

- collection of python files that contains **re-usable functions**, which can be imported to other files.
- Collection of such modules is known as **package**

## 10. MULTIPLE IMPORT STATEMENTS

Importing entire module	<code>import math math.sqrt(10)</code>
Importing using different Alias names	<code>import math as m m.sqrt(10)</code>
Importing only required functions/classes	<code>from math import sqrt sqrt(10)</code>
Import everything within module	<code>from math import * sqrt(10)</code>

## 11. EXCEPTION HANDLING

- Python known error with cause are known as **exceptions**
- Program gets terminated as an exception occurs.

## TRY-EXCEPT

mechanism of handle exceptions	<code>try:</code> <code># code that may cause exception</code> <code>except:</code> <code># what to do when exception occurs</code>
handling specific exceptions	<code>try:</code> <code>return a / 0</code> <code>print(5 + 4)</code> <code># any amount of code</code> <code>except ZeroDivisionError:</code> <code>print("WHY ARE YOU DIVIDING BY ZERO?")</code>
<b>finally</b> block runs even if an exception occurs or not and free all the allocated resources if any.	<code>try:</code> <code>print("I am trying!")</code> <code>1/0</code> <code>except:</code> <code>print("Except")</code> <code>finally:</code> <code>print("FINALLLYYYY!!")</code>

## 12. CUSTOM EXCEPTIONS

raising custom exceptions	<code>raise Exception("custom exception")</code>
creating custom exceptions: Just inherit the base class of <b>Exception</b> and add req functionalities	<code>class MyCustomException(Exception):     pass</code>