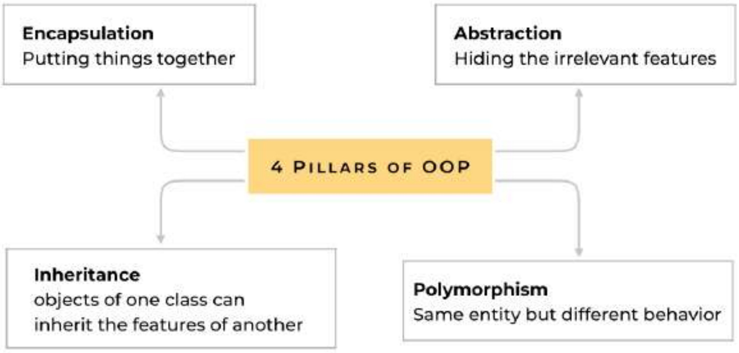
PYTHON

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")
```

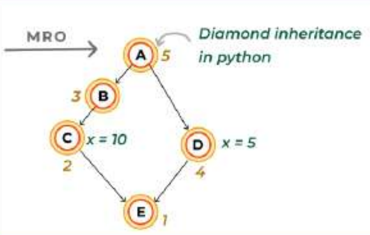
- Objects of one class can inherit the features of another

```
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
```

Using decorator on other function

```
@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

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

krwargs are variable size keyword arguments stored inside dictionary

```python
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	r	Read and Write	r+	Read binary	rb	Append and Read	a+
Write only	w	Write and Read	w+	Write binary	wb	Append Only	a

Working with Files

Opening a file

```python
file = open("sample.txt", "r")
```

Closing a file

```python
file.close()
```

Writing to a file

```python
file = open("sample.txt", "w+")
file.writelines(["1\n", "2\n"])
# write from list
file.write(“hellow world”)
# write from string
file.close()
```

Reading from a file

```python
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()
```

Reading large files

```python
file = open("sample2.txt", "r+")
buffer = file.readline()
while buffer:
# n lines = 1 block of memory
   print(buffer, end = "")
   buffer = file.readline()
file.close()
```

Moving reading/ writing cursor

```python
file.seek(3)
# moving 3 characters ahead
```

Smart way of working with files.

“**with**” statement simplifies exception handling by encapsulating common preparation and cleanup tasks."

```python
with open("sample3.txt", "r+") as file:
   print(file.read(5))
   file.seek(0)
   print(file.read())
```

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	`import math`
	`math.sqrt(10)`

Importing using different Alias names	`import math as m`
	`m.sqrt(10)`

Importing only required functions/classes	`from math import sqrt`
	`sqrt(10)`

Import everything within module	`from math import *`
	`sqrt(10)`

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

```python
try:
 # code that may cause exception
except:
 # what to do when exception occurs
```

handling specific exceptions

```python
try:
   return a / 0
   print(5 + 4)
   # any amount of code
except ZeroDivisionError:
   print("WHY ARE YOU DIVIDING BY ZERO?")
```

`finally` block runs even if an exception occurs or not and free all the allocated resources if any.

```python
try:
   print("I am trying!")
   1/0
except:
   print("Except")
finally:
   print("FINALLYYY!!")
```

12. Custom Exceptions

raising custom exceptions

```python
raise Exception("custom exception")
```

creating custom exceptions: Just inherit the base class of `Exception` and add req functionalities

```python
class MyCustomException(Exception):
    pass
```