La POO puede variar según el programador.

Este tipo de programación es **mucho más abierta**, aunque **favorece** una **estructuración ordenada**. Se **requiere** de una **cierta formación** previa, pero en la **práctica** hay varias **ventajas**. La organización del código se realiza en distintas clases que, posteriormente, podrán concretarse en objetos.

La **POO busca** que las **aplicaciones** que se **desarrollen** sean **cada vez más complejas sin** que eso suponga **desechar** el **código**. Esta filosofía permitirá reutilizarlo, de manera que progresar no supondrá renunciar.

Herencia:

La **herencia** es un proceso mediante el cual se puede **crear** una **clase hija** que **hereda de** una **clase padre**, **compartiendo** sus **métodos** y **atributos**.

```python
#POO Python
#Herencia. Herencia múltiple. Sobreescritura. Constructor super
class Padre: #clase Base, Padre, SuperClase
    color='red'
    def __init__(self,categoria) -> None:
        self.categoria=categoria

    @staticmethod
    def consultar():
        print('resultado de clase Padre')

class General:
    color='blue'

#Python soporta la herencia múltiple en orden de las comas
class Hija(Padre,General): #clase Derivada, Hija, Subclase
    unidades=20
    def __init__(self,precio,categoria) -> None:
        super().__init__(categoria) #sin puntos.
        self.precio=precio

    @staticmethod
    def consultar():
        print(f'Consultando desde la hija')

hija1=Hija(19.95,'oficial') #instanciar la clase
print(f'El total es {hija1.unidades*hija1.precio}')
print(f'El color elegido es: {hija1.color}') #usar atributos de clase Base
hija1.consultar() #Usar métodos de clase Base
```

Polimorfismo:

El término **polimorfismo** tiene origen en las palabras ***poly* (muchos)** y ***morfo* (formas)**, y aplicado a la programación hace referencia a que los **objetos** pueden **tomar diferentes formas**.

Objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un **comportamiento distinto** (tomando diferentes formas) según cómo sean accedidos.

```python
#Polimorfismo con anulación de métodos y crear lista de animales

from types import CoroutineType
from unicodedata import name


class Animal(): #clase base
    @staticmethod
    def comer():
        print('un animal come')

class Perro(Animal):
    @staticmethod
    def comer(): #anulación de método
        print('El PERRO come pienso para perros')

class Gato(Animal):
    @staticmethod
    def comer():
        print('El GATO come pienso para gatos')
perro=Perro()
gato=Gato()

perro1=Perro()
perro2=Perro()
perro3=Perro()
perro4=Perro()
gato1=Gato()
gato2=Gato()
gato3=Gato()

animales=[gato1,perro2,gato2,perro3,gato3] #lista
for animal in animales:
    print(animales)
    animal.comer() #polimorfismo
```

Encapsulamiento:

El **encapsulamiento** o encapsulación en programación es un concepto relacionado con la programación orientada a objetos, y hace referencia al **ocultamiento** de los **estados internos** de una clase **al exterior**. Dicho de otra manera, **encapsular consiste en hacer que los atributos o métodos internos a una clase no se puedan acceder ni modificar desde fuera**, sino que tan solo el propio objeto pueda acceder a ellos.

```python
#POO Python. Encapsulamiento
#https://ellibrodepython.com/encapsulamiento-poo

class Cliente:
    __descuento=True #atributo privado por el doble __
    def __init__(self,nombre) -> None: #constructor
        self.nombre=nombre
    def __saludar(self): #método private
        print(f'Hola {self.nombre}')
    def despedir(self):
        print(f'Adiós {self.nombre}')

cliente1=Cliente('Juan López') #instanciar
#cliente1.__saludar  Da error porque es privado :)
#los atributos y métodos privados sólo se pueden utilizar dentro de la misma clase
```

Sobreescritura:

El término **sobreescritura** de funciones o métodos es comúnmente **usado en la herencia de la POO**. Consiste en **instanciar en una clase base** algún **método** los cuales **serán sobreescritos en la clase derivadas** con el **mismo nombre** del método **usado en la clase base**, de esta forma el **método** llamado **dependerá** del **tipo** del **objeto** que **llame** al **método** es importante resaltar que estos métodos sobreescritos pertenecen a la interfaz de ambas clases y **el método en la clase derivada sobreescribe al de la clase base**.

En la imagen se ve un ejemplo de sobreescritura en la línea 45 y 58. Donde la clase padre tiene definido el método consultar() y lo sobreescribimos en la clase hija con el mismo nombre consultar()

```
36  #POO Python
37  #Herencia. Herencia múltiple. Sobreescritura. Constructor super
38  class Padre: #clase Base, Padre, SuperClase
39      color='red'
40      def __init__(self,categoria) -> None:
41          self.categoria=categoria
42
43      @staticmethod
44      def consultar():
45          print('resultado de clase Padre')
46
47  class General:
48      color='blue'
49
50  #Python soporta la herencia múltiple en orden de las comas
51  class Hija(Padre,General): #clase Derivada, Hija, Subclase
52      unidades=20
53      def __init__(self,precio,categoria) -> None:
54          super().__init__(categoria) #sin puntos.
55          self.precio=precio
56
57      @staticmethod
58      def consultar():
59          print(f'Consultando desde la hija')
60
61  hija1=Hija(19.95,'oficial') #instanciar la clase
62  print(f'El total es {hija1.unidades*hija1.precio}')
63  print(f'El color elegido es: {hija1.color}') #usar atributos de clase Base
64  hija1.consultar() #Usar métodos de clase Base
65
```

Sobrecarga:

La **sobrecarga** de Métodos **se apoya de métodos y constructores**.

La sobrecarga de métodos **hace** que **un mismo nombre** pueda **representar distintos métodos** con **distinto tipo y número de parámetros**, manejados dentro de la misma clase. La sobrecarga de métodos **se refiere a la posibilidad de tener dos o más métodos con el mismo nombre**, **pero distinta funcionalidad.** Es decir, dos o más métodos con el mismo nombre realizan acciones diferentes y el compilador usará una u otra dependiendo de los parámetros usados. Esto también se aplica a los constructores (de hecho, es la aplicación más habitual de la sobrecarga).

```python
#Crear una clase llamada Operaciones
#El método sumar me permite sumar dos números o tres
#crea un objeto de operaciones y prueba los dos métodos

from ipaddress import summarize_address_range


class Operaciones:
    def __init__(self) -> None:
        pass
    def sumar(self,a,b,c=None): #Sobrecaarga
        if c is None:
            return a+b
        else:
            return a+b+c

operacion=Operaciones()
resultado1=operacion.sumar(4,5)
print(resultado1)
resultado2=operacion.sumar(1,4,6)
print(resultado2)

#no es necesario clases
#por consola pides la fecha del examen
#introduces la fecha correcta... que es 14/12/2022
#te muestran los días que faltan

from datetime import datetime

def dias_restantes():
    fecha=input('Dime la fecha del examen: ')
    dia_examen=datetime.strptime(fecha,'%d-%m-%Y')
    hoy=datetime.now()
    dias=dia_examen-hoy
    print(dias)
dias_restantes()

```

Modificadores de acceso(public / private):

La mayoría de los lenguajes de programación tienen **tres formas** de modificadores de acceso, que
son **Público** , **Protegido** y **Privado** en una clase. En el caso de **Python solo** usamos **Público y Privado**.

Python usa el símbolo '_' para determinar el control de acceso para un miembro de datos específico o una función miembro de una clase. Los especificadores de acceso en Python tienen un papel importante que desempeñar para proteger los datos del acceso no autorizado y evitar que sean explotados.

Se ve mejor con un ejemplo visual:

En este ejemplo de encapsulamiento se ve como utilizamos la '_' para marcar como privado el descuento del cliente en la línea 114 o en la 117 en saluda.

```
110  #POO Python. Encapsulamiento
111  #https://ellibrodepython.com/encapsulamiento-poo
112
113  class Cliente:
114      __descuento=True #atributo privado por el doble __
115      def __init__(self,nombre) -> None: #constructor
116          self.nombre=nombre
117      def __saludar(self): #método private
118          print(f'Hola {self.nombre}')
119      def despedir(self):
120          print(f'Adiós {self.nombre}')
121
122  cliente1=Cliente('Juan López') #instanciar
123  #cliente1.__saludar  Da error porque es privado :)
124  #los atributos y métodos privados sólo se pueden utilizar dentro de la misma clase
```

```
__descuento=True #atributo privado por el doble __
```

```
def __saludar(self): #método private
```