Python:

Clases, Objetos, Atributos, Métodos

Una **Clase** es una forma de empaquetar atributos y métodos de algo que podríamos representar en el mundo real. Por ejemplo una clase puede ser los estudiantes o las profesiones, animales.

Un **Objeto** es una instancia de una **clase**. Por ejemplo la clase pueden ser animales y un objeto sería un pero. Si la clase fueran aparatos electrónicos un objeto podría ser un ordenador.

Los **Atributos** son descripciones de una característica. De un perro, una de sus características podría ser el color, su peso, su raza.

Un **Método** son acciones que un **objeto** o **clase** pueden usar. Una acción o método podría ser correr, comer, ladrar.

Para definir una clase, sólo tenemos que poner class, seguido de una palabra que empieza en mayúsculas. Toda clase necesita definir un **constructor**, __init__ que es el método que se llama para crear un objeto de esa clase. (dos subguiones, init y dos subguiones más). self hace referencia a la misma clase. La realidad es que el método __init__ crea el objeto y luego lo inicializa, no es el constructor como tal, en cambio el método **__new__** sólo construye el objeto. Utilizaremos __init__

```
class Trabajador:            #Definición de una clase

        def __init__(self, nombre, edad):
                self.nombre = nombre
                self.edad = edad

        #setters   actualizan un valor (atributo) dentro de la clase
        def set_salario(self, salario):      #siempre debemos de colocar
                self.salario = salario       #al menos un argumento en cada
                                             #función que creamos de una clase.
        #getters
        def get_salario_neto(self):
                return self.salario - 0.10 * self.salario

#Instancia de un objeto de esa clase
trabajador_uno = Trabajador('Juan', 40)    #NO hace falta llamar a init
#print(trabajador_uno)

print(trabajador_uno.nombre)
print(trabajador_uno.edad)

trabajador_uno.set_salario(400)

print(trabajador_uno.salario)
print(trabajador_uno.get_salario_neto())

#Actualizar datos. NO es necesario llamar a métodos, se hace directamente
trabajador_uno.nombre = 'Pedro'
print(trabajador_uno.nombre)
trabajador_uno.edad = 35
print(trabajador_uno.edad)
```

Herencia

La **Herencia** nos va a servir para crear **clases** nuevas a partir de **clases** existentes.

```
class Doctor(Trabajador):

        def __init__(self, nombre, edad, especialidad):
                Trabajador.__init__(self, nombre, edad)
                self.especialidad = especialidad

        def horario_operacion(self):
                if self.especialidad == 'cardiólogo':
                        return '8 am'
                elif self.especialidad == 'neurólogo':
                        return '4 pm'
                else:
                        return '12 pm'
        def get_salario_neto(self):                  #modificar los métodos del padre.
                Trabajador.set_salario(self, 1000)
                if self.especialidad == 'cardiólogo':
                        return self.salario * 1.5
                elif self.especialidad == 'neurólogo':
                        return self.salario * 2
                else:
                        return self.salario

doctor = Doctor('Luis', 45, 'cardiólogo')
print(doctor.edad)
print(doctor.horario_operacion())
print(doctor.get_salario_neto())
```

```
#Herencia MÚLTIPLE:
class Clase1:
        pass
class Clase2:
        pass
class Clase3(Clase1, Clase2):
        pass

print(Clase3.__mro__)

#Method Order Resolution. El orden de búsqueda de los métodos. Propia clase, subiendo, clase padre
de izquierda a derecha. Supper() permite acceder a los métodos de la clase padre.
#todas heredan de la clase object aunque NO lo especifiquemos.

#Métodos de clase @classmethod
#Métodos estáticos @staticmethod
class Clase:
        def metodo(self):
                return 'Método normal', self

        @classmethod
        def metododeclase(cls):       #no pueden acceder a los atributos de la instancia si de la
clase incluso modificar.
                return 'Método de clase', cls

        @staticmethod
        def metodoestatico():         #no aceptan parámetros.
                return "Método estático"




##Otro ejemplo:       también variables como atributos de clase

#Se crea la clase ciudad que hereda de object.

class Ciudad(object):
        #Datos
        cont_ciudad = 0
        id_ciudad = 0

        def __init__(self, nombre='', x=0, y=0):
                """Constructor que recibe el nombre de la ciudad y sus coordenadas x e y"""
                self.nombre = nombre

                self.x = x
                self.y = y

                Ciudad.cont_ciudad += 1

                self.id_ciudad = Ciudad.cont_ciudad


        def __str__(self):
                """Método que retorna un string con la info de la ciudad"""
                return 'Ciudad: ' + self.nombre + ',id= ' + str(self.id_ciudad) + ',x= %s,y=%s' %
(self.x,self.y)



        def __set__( self, nombre):
                """Método que asigna un valor"""
                self.nombre= nombre



        def __get__( self):
                """Método que obtiene un valor"""
                return  self.nombre



        def mover_a(self,x=0,y=0):
                """Método que cambia de valor a x e y"""
                self.x += x
                self.y += y



        def distancia(self, otra_ciudad):
                """Método que calcula la distancia con respecto a otra ciudad"""

                xi = pow(otra_ciudad.x-self.x,2)
                yi = pow(otra_ciudad.y-self.y,2)

                return sqrt(xi+ yi)
```

```
        def __del__(self):
                """Elimina la clase"""
                #obtener el nombre de una clase
                class_name = self.__class__.__name__

                print('class ', class_name, 'destroyed')


        #Inicializando un Método estatico
        #Es el caso que no se referencia al objeto en si mismo con
        #self

        @staticmethod
        def info():
                """Método que devuelve el nombre de la asignatura"""
                return  "PIA-- Python OO."



if __name__ == "__main__":
        a = Ciudad('Valencia',5,5)
        b = Ciudad('Maracay',5,15)

        print(a)
        print(b)
        print(Ciudad.cont_ciudad)
        a.mover_a(4,3)
        b.mover_a(7,12)
        print(a.info())
        print(a)
        print(b)

##OJO, prueba esto:
class Servicio():
        dato = []

        def __init__(self,otro_dato):
                self.otro_dato = otro_dato


s1 = Servicio(["a","b"])
s2 = Servicio(["c","d"])

s1.dato.append(1)

print("dato de S1: ",s1.dato)
print("dato de S2: ",s2.dato)

s2.dato.append(2)

print("dato de s1: ",s1.dato)
print("dato de s2: ",s2.dato)

print("otro dato de s1: ", s1.otro_dato)
print("otro dato de s2: ", s2.otro_dato)


#SOLUCIÓN      dato = None
class Servicio():
        dato = None

        def __init__(self,otro_dato):
                self.otro_dato = otro_dato

s1 = Servicio(["a","b"])
s2 = Servicio(["c","d"])

s1.dato = 1

print("dato de S1: ",s1.dato)
print("dato de S2: ",s2.dato)

s2.dato = 2

print("dato de s1: ",s1.dato)
print("dato de s2: ",s2.dato)



#Ahora sin la instancia de la clase, creando dato como una variable privada y unos métodos que
acceden a dato:
class Servicio():
        def __init__(self,otro_dato):
                self.__dato = []
                self.otro_dato = otro_dato

        def mostrar_dato(self):
                return self.__dato
```

```
        def agregar_dato(self,dato):
                self.__dato.append(dato)

        def inicializar_dato(self,):
                self.__dato = []

s1 = Servicio(["a","b"])
s2 = Servicio(["c","d"])
try:
        s1.__dato.append(1)
except AttributeError:
        print("No se pudo agregar a dato de s1")
finally:
        print(s1.mostrar_dato())
        print(s2.mostrar_dato())

        s1.agregar_dato(1)
        s1.agregar_dato(3)
        s2.agregar_dato(2)
        s2.agregar_dato(4)

        print(s1.mostrar_dato())
        print(s2.mostrar_dato())
```

Uno de los *decorators* más interesantes que podemos utilizar es @property, que nos permite definir métodos en una clase para consultar y modificar un atributo interno.

Puede ser usado sobre un método para que actúe como si fuera un atributo, permite encapsular:

```
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo

    @property
    def mi_atributo(self):
        return self.__mi_atributo


mi_clase = Clase("valor_atributo")
mi_clase.mi_atributo          #NO puede ser llamado con ()



#otro ejemplo:
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo   #lo ocultamos con __.

mi_clase = Clase("valor_atributo")

#NO podemos acceder a:  mi_clase.__mi_atributo   da error.



#otro ejemplo:   @property en el setter (modificar el contenido)
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo

    @property
    def mi_atributo(self):
        return self.__mi_atributo

    @mi_atributo.setter
    def mi_atributo(self, valor):
        if valor != "":
            print("Modificando el valor")
            self.__mi_atributo = valor
        else:
            print("Error está vacío")


mi_clase = Clase("valor_atributo")
mi_clase.mi_atributo

mi_clase.mi_atributo = "nuevo_valor"
mi_clase.mi_atributo

mi_clase.mi_atributo = ""
```