TEMA IV

DESARROLLO DE APLICACIONES DE PROPÓSITO GENERAL

"Siempre parece imposible,
hasta que se hace"
Nelson Mandela

1. PROGRAMACIÓN ORIENTADA A OBJETOS

Python implementa la orientación a objetos.

Clases y objetos

Una clase define los datos y la lógica de un objeto. La lógica se divide en funciones: métodos y variables: propiedades o atributos. En python tenemos atributos de case y atributos de objeto

```
class nombre():
    atributos de la clase (se inicializan aquí)
    def __init__(self, atributos de objeto):
    def  método(self,parámetros):
```

Un atributo de clase es compartido por todas las instancias de una clase. Se definen dentro de la clase (después del encabezado de la clase) pero nunca dentro de un método. Este tipo de atributos son conocidos en otros lenguajes como variables estáticas.

Un atributo de clase puede ser accedido con el nombre de la clase(como en otros lenguajes) pero también con una instancia.

Un atributo de objeto se define dentro de un método(generalmente el constructor) y pertenece a un objeto determinado de la clase instanciada.

Tenemos un método especial, llamado constructor __init__, que nos permite inicializar los atributos de objetos. Este método se llama cada vez que se crea una nueva instancia de la clase.

El método constructor, al igual que todos los métodos de cualquier clase, recibe como primer parámetro a la instancia sobre la que está trabajando. Por convención a ese primer parámetro se lo suele llamar self (que podríamos traducir como yo mismo), pero puede llamarse de cualquier forma.

Para referirse a los atributos de objetos hay que hacerlo a partir del objeto self.

Ejemplo: Implementa una clase Tarjeta con dos atributos de objeto cantidad e identificador y un atributo de clase comisión. Define métodos para sacar dinero, meter dinero y ver el saldo. Cada vez que sacamos dinero cobramos una comisión (en todas las tarjetas la misma). La cuenta se abre con una cantidad mínima de 1000€.

```python
class Tarjeta():
    comsion=0.5
    def __init__(self,identificador,cantidad=0):
        self.identificador=identificador
        self.cantidad=cantidad

    def sacar(self,cantidad):
        self.cantidad=self.cantidad-cantidad-Tarjeta.comsion
      # self.cantidad=self.cantidad-cantidad-self.comsion

    def ingresar(self,cantidad):
        self.cantidad+=cantidad

    def verSaldo(self):
        return self.cantidad
```

En otro archivo implementamos un script para probarla

```python
from Tarjeta import Tarjeta
t=Tarjeta("111111abc",1000)
print(t.verSaldo())
t.sacar(500)
print(t.verSaldo())
Tarjeta.comsion=0.25
```

Ámbito de los atributos de objeto

En Python, los atributos de los objetos son por defecto públicos y por tanto accesibles desde fuera de la clase. Para que no se vean desde fuera, y por tanto sean privados, su nombre debe empezar por __ (dos subrayados). Para determinar que un atributo es protected usaremos un sólo subrayado _ .

```
class Tarjeta():
    comsion=0.5
    def __init__(self,identificador,cantidad=0):
        self.__identificador=identificador
        self.__cantidad=cantidad

    def sacar(self,cantidad):
        self.__cantidad=self.__cantidad-cantidad-Tarjeta.comsion
      # self.__cantidad=self.__cantidad-cantidad-self.comsion

    def ingresar(self,cantidad):
        self.__cantidad+=cantidad

    def verSaldo(self):
        return self.__cantidad
```

De igual forma, si queremos que un método tenga ámbito privado, lo declaramos de forma análoga: def __nombre()

Si los defino privados y los usamos fuera de la clase, el intérprete no da error pero realmente no cambia el valor de la variable del objeto.

```
from Tarjeta import Tarjeta
t=Tarjeta("111111abc",1000)
print(t.verSaldo())
t.__cantidad=5
t.cantidad=7
t.sacar(500)
print(t.verSaldo())
Tarjeta.comsion=0.25
```

Ahora bien, python realmente no "encubre" el valor de los atributos, no es como Java o C#. Cada miembro con __ se cambiará a object._class__variable por lo que desde fuera de la clase podemos acceder de esa manera a cualquier atributo privado. Pasa lo mismo con los protected, son directamente accesibles desde fuera de la clase. No tienen mucho sentido declararlos con un sólo _ , a menos que queramos seguir las directrices de otros lenguajes y no los usemos directamente desde fuera de la clase. Pero tenemos que ser conscientes de que cualquiera que use mi clase podrá usar esos atributos directamente desde sus scripts.

```
print(t._Tarjeta__cantidad)
```

Debido a esto, hay programadores que por sistema declaran todo público, no es buena práctica desde mi punto de vista. Creo que hay que declararlos private y usarlos de la forma adecuada tanto si yo he diseñado mis clases como si utilizo las de otro programador

Propiedades Getter, Setter y Deleter

En Python para definir los métodos:

- get utilizamos el decorador @property
- set utilizamos el decorador @nombredelapropiedad.setter
- deleter utilizamos el decorador @nombredelapropiedad.deleter para borrar el atributo, no su contenido sino la variable.

Ejemplo: Añadimos a nuestra clase Tarjeta estos métodos decorados. Para el identificador sólo generamos el getter.

```python
class Tarjeta():
    comsion=0.5
    def __init__(self,identificador,cantidad=0):
        self.__identificador=identificador
        self.__cantidad=cantidad

    @property
    def cantidad(self):
        return self.__cantidad

    @cantidad.setter
    def cantidad(self,valor):
        self.__cantidad=valor

    @property
    def identificador(self):
        return self.__identificador
```

Métodos especiales para sobreescribir

Función str

Esta función nos sirve para convertir números en cadenas. Podemos utilizarla en una clase para pasarle a la función print lo que queremos escribir de nuestros objetos (análogo al toString de java)

```python
def __str__(self):
    cadena=type(self).__name__+"\n"
    cadena=cadena+"Identificador: "+str(self.__identificador)+" - "+"Saldo: "+str(self.__cantidad)
    return cadena
```

Para utilizarla sólo tengo que pasarle un objeto a la función print, ésta a su vez llama a __str__

```
from Tarjeta import Tarjeta
t=Tarjeta("111111abc",1000)
print(t)
```

Igualdad de objetos

Los objetos en Python se tratan como referencias, por lo tanto, si declaro dos objetos en dos variables aunque tengan los mismos valores el operador == me dirá que son diferentes.

Para comparar objetos por sus datos almacenados tengo que implementar la función __eq__

```
def __eq__(self, otraTarjeta):
    return self.__identificador==otraTarjeta.identificador and self.__cantidad==otraTarjeta.cantidad
```

```
from Tarjeta import Tarjeta
t=Tarjeta("111111abc",1000)
q=Tarjeta("111111abc",1000)
if t==q:
    print ("si")
else:
    print("no")
```

Suma y resta de objetos

Podemos sumar y restar objetos utilizando los operadores + y – si implementamos en nuestra clase los métodos __add__ y __sub__ (En lenguajes como C++ ésto se llama sobrecarga de operadores)

```
def __add__(self, otraTarjeta):
    self._cantidad=self._cantidad+otraTarjeta._cantidad
    otraTarjeta._cantidad=0
```

Por ejemplo en nuestro ejercicio de las Tarjetas, queremos poder sumar dos tarjetas porque vamos a pasar el saldo de una a otra

```
t=Tarjeta("111111abc")
q=Tarjeta("aaa",100)
t+q
print(t)
print(q)
```

Hay que tener en cuenta que estos métodos no retornan una tarjeta, cambian el valor de la tarjeta que ponemos en primer lugar

Por ejemplo, si quiero hacerme una tarjeta nueva con los saldos de otras dos tarjetas que ya tengo

```
from tarjeta import Tarjeta

t=Tarjeta("111111abc")
q=Tarjeta("aaa",2000)
print(t.cantidad)
t+q

h=Tarjeta("nueva",t.cantidad)

print(h)
```

Hay otros métodos especiales que se pueden sobreescribir

Polimorfismo, Herencia y Delegados

El polimorfismo es la técnica que nos posibilita que, al invocar un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Esto se debe a que distintos objetos pueden tener un método con un mismo nombre, pero que realice distintas operaciones.

Lo llevamos usando desde principio del curso, por ejemplo, podemos recorrer con una estructura for distintas clases de objeto, debido a que el método especial __iter__ está definida en cada una de las clases. Otro ejemplo sería que con la función print podemos imprimir distintas clases de objeto, en este caso, el método especial __str__ está definido en todas las clases.

La herencia es un mecanismo de la programación orientada a objetos que sirve para crear clases nuevas a partir de clases preexistentes. Se toman (heredan) atributos y métodos de las clases viejas y se los modifica para modelar una nueva situación.

La clase desde la que se hereda se llama clase base y la que se construye a partir de ella es una clase derivada.

class nombre(clase padre):

atributos de la clase (se inicializan aquí)

def __init__(self, atributos de objeto):

def método(self,parámetros):

La función super nos proporciona una referencia a la clase base (padre)

Podemos sobreescribir métodos

Ejemplo: “Añadimos una clase Tjoven, tarjetas para clientes de menos de 26 años, con una cantidad mínima de 200€. Sin comisión al sacar dinero y un regalo de 10€ por cada cumpleaños hasta los 22 años.

```python
from tarjeta import Tarjeta
class TJoven(Tarjeta):
    #CONSTRUCTOR si no pasamos por defecto la edad, no puede ir dentras de cantidad al no ser por defecto
    """
    def __init__(self,edad,identificador,cantidad=200):
        super().__init__(identificador,cantidad)
        self.__edad=edad
    """
    #CONTRUCTOR si pasamos la edad máxima por defecto
    def __init__(self,identificador,cantidad=200,edad=25):
        super().__init__(identificador,cantidad)
        self.__edad=edad

    #PROPIEDADES GETTER Y SETTER
    @property
    def edad(self):
        return self.__edad
    @edad.setter
    def edad(self,value):
        self.__edad=value

    #MÉTODOS ESPECIALES
    def __str__(self):
        cadena=super().__str__() + " - Edad: "+str(self.__edad)
        return cadena

    #MÉTODOS DE LA CLASE
    def cumpleAnyos(self):
        if (self.__edad<22):
            self.__cantidad+=10

    def sacar(self,value):
            #self.cantidad=self.cantidad-value
            self._Tarjeta__cantidad-=value
```

Funciones issubclass() y isinstance()

La función issubclass(SubClase, ClaseSup) se utiliza para comprobar si una clase (SubClase) es hija de otra superior (ClaseSup), devolviendo True o False según sea el caso.

La función booleana isinstance(Objeto, Clase) se utiliza para comprobar si un objeto pertenece a una clase o clase superior. Cualquier objeto de será de su tipo y del de su padre.

Herencia múltiple

En Python es posible la herencia múltiple, es decir una clase puede heredar de varias. “No tiene un único padre”. Es muy importante nombrar bien los atributos y métodos de cada clase para no generar conflictos

```
class Telefono:
    "Clase teléfono"
    def __init__(self,numero):
        self.__numero=numero
    def telefonear(self):
        print('llamando')
    def colgar(self):
        print('colgando')
    def __str__(self):
        return self.__numero

class Camara:
    "Clase camara fotográfica"
    def __init__(self,mpx):
        self.__mpx=mpx
    def fotografiar(self):
        print('fotografiando')
    def __str__(self):
        return str(self.__mpx )+ "Mpx"
class Reproductor:
    "Clase Reproductor Mp3"
    def __init__(self,capcidad):
        self.__capacidad=capcidad
    def reproducirmp3(self):
        print('reproduciendo mp3')
    def reproducirvideo(self):
        print('reproduciendo video')
    def __str__(self):
        return str(self.__capacidad)    +"G"

class Movil(Telefono, Camara, Reproductor):
    def __init__(self,numero,mpx,capacidad):
        Telefono.__init__(self,numero)
        Camara.__init__(self,mpx)
        Reproductor.__init__(self,capacidad)
    def __str__(self):
        return "Número: {0}, Cámara: {1},Capacidad: {2}".format(Telefono.__str__(self),Camara.__str__(self),Reproductor.__str__(self))
```

Al heredar de varias clases, python hereda por defecto el constructor de la primera clase que pongamos entre los paréntesis. Así mismo para métodos con el mismo nombre también se ejecutarán los métodos de la primea clase

```
mo=Movil(1111,5,1)
print(mo)
```