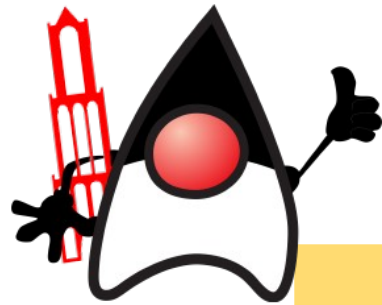
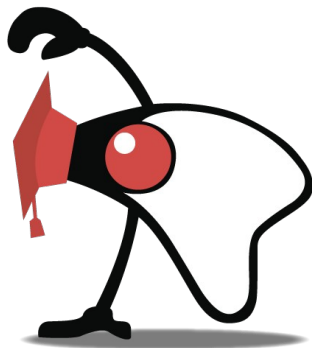
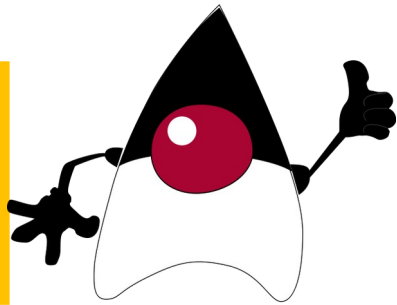




Curso FullStack Python

Codo a Codo 4.0



Python

Parte 6



Colaboración de clases

Normalmente en un problema resuelto con la metodología de programación orientada a objetos no interviene una sola clase, sino que hay **muchas clases que interactúan** y se comunican. Plantearemos un problema separando las actividades en dos clases.

Problema 7:

Un banco tiene 3 clientes que pueden hacer depósitos y extracciones. También el banco requiere que al final del día calcule la cantidad de dinero que hay depositado.

Lo primero que hacemos es identificar las clases: Cliente y Banco, luego debemos definir los atributos y los métodos de cada clase:

Cliente

atributos

nombre

monto

métodos

__init__

depositar

extraer

retornar_monto

Banco

atributos

3 Cliente (3 objetos de la clase Cliente)

métodos

__init__

operar

depositos_totales



Primero hacemos la declaración de la clase **Cliente**, en el método `__init__` inicializamos los atributos nombre con el valor que llega como parámetro y el atributo monto con el valor cero. Recordemos que en Python para diferenciar un atributo de una variable local o un parámetro le antecedemos la palabra clave `self` (es decir nombre es el parámetro y `self.nombre` es el atributo):

```
class Cliente:

    def __init__(self,nombre):
        self.nombre=nombre
        self.monto=0
```

El resto de los métodos de la clase Cliente quedará de la siguiente manera

```
def depositar(self,monto):
    self.monto=self.monto+monto

def extraer(self,monto):
    self.monto=self.monto-monto

def retornar_monto(self):
    return self.monto

def imprimir(self):
    print("{} tiene depositada la suma de {}".format(self.nombre,self.monto))
```

En el bloque principal no se requiere crear objetos de la clase Cliente, esto debido a que los clientes son atributos del Banco.

Luego creamos la clase Banco, que tendrá 3 objetos de la clase Cliente:

```
def __init__(self):  
    self.cliente1=Cliente("Juan")  
    self.cliente2=Cliente("Ana")  
    self.cliente3=Cliente("Diego")  
)
```

Con el método **operar()** llamamos a algunos métodos de la clase Cliente: los 3 clientes depositaron diferentes montos y uno de ellos realizó una extracción:

```
def operar(self):  
    self.cliente1.depositar(100)  
    self.cliente2.depositar(150)  
    self.cliente3.depositar(200)  
    self.cliente3.extraer(150)
```

Con el método **depositos_totales()** sumamos los montos de los 3 clientes, los guardamos en la variable **total** que imprimiremos y llamamos al método **imprimir** de la clase Cliente:

```
def depositos_totales(self):  
    total=self.cliente1.retornar_monto()+self.cliente2.retornar_monto()  
+self.cliente3.retornar_monto()  
    print("El total de dinero del banco es: {}".format(total))  
    self.cliente1.imprimir()  
    self.cliente2.imprimir()  
    self.cliente3.imprimir()
```

En el programa principal creamos el objeto Banco y llamaremos a los métodos **operar()** y **depositos_totales()**:

```
banco1=Banco()  
banco1.operar()  
banco1.depositos_totales  
( )
```

```
El total de dinero del banco es: 300  
Juan tiene depositada la suma de 100  
Ana tiene depositada la suma de 150  
Diego tiene depositada la suma de 50
```

terminal



*Ejercicio_7_POO.p
y*

¿Cómo es, entonces, el flujo del programa?

1. Se crea el objeto de tipo Banco que tendrá 3 clientes: Juan, Ana y Diego.
2. Al llamar al método operar() de la clase Banco se llama a los métodos depositar() y extraer() de la clase Cliente.
3. Al llamar al método depositos_totales() de la clase Banco se llama al método retornar_monto() de la clase Cliente y al método imprimir() de la misma clase que mostrará el nombre y lo que tiene depositado.

Colaboración de clases

Problema 8:

Plantear un programa que permita jugar a los dados. Las reglas de juego son: se tiran tres dados y si los tres salen con el mismo valor se debe mostrar un mensaje que diga "ganó", sino "perdió".

Lo primero que hacemos es identificar las clases: Dado y JuegoDeDados, luego debemos definir los atributos y los métodos de cada clase:

Dado

atributos

valor

métodos

tirar

imprimir

retornar_valor

JuegoDeDados

atributos

3 Dado (3 objetos de la clase Dado)

métodos

__init__

jugar



Importamos el módulo "random" de la biblioteca estándar de Python ya que requerimos utilizar la función randint: **import random**

La clase Dado define un método tirar que almacena en el atributo valor un número aleatorio comprendido entre 1 y 6.

Los otros dos métodos de la clase Dado tienen por objetivo mostrar el valor del dado y retornar dicho valor a otra clase que lo requiera.

```
class Dado:

    def tirar(self):
        self.valor=random.randint(1,6)

    def imprimir(self):
        print("Valor del dado: {}".format(self.valor)
)

    def retornar_valor(self):
        return self.valor
```


La clase JuegoDeDados define tres atributos de la clase Dado, en el método `__init__` crea dichos objetos:

```
class JuegoDeDados:

    def __init__(self):
        self.dado1=Dado()
        self.dado2=Dado()
        self.dado3=Dado()

    def jugar(self):
        self.dado1.tirar()
        self.dado1.imprimir()
        self.dado2.tirar()
        self.dado2.imprimir()
        self.dado3.tirar()
        self.dado3.imprimir()
        if self.dado1.retornar_valor()==self.dado2.retornar_valor(
) and self.dado1.retornar_valor()==self.dado3.retornar_valor():
            print("Ganó")
        else:
            print("Perdió")
```

En el bloque principal se crea el objeto JuegoDeDados y se llama al método jugar() del mismo método:

```
juego_dados=JuegoDeDados(  
)  
juego_dados.jugar()
```

terminal
Valor del dado: 3
Valor del dado: 2
Valor del dado: 6
Perdió

terminal
Valor del dado: 3
Valor del dado: 3
Valor del dado: 3
Ganó



*Ejercicio_8_POO.p
y*

¿Cómo es, entonces, el flujo del programa?

1. Se crea el objeto de tipo JuegoDeDados que tendrá 3 dados (objetos).
2. Al llamar al método jugar() de la clase JuegoDeDados se llama a los métodos tirar() e imprimir() de la clase Dado. En el primer caso se genera un número aleatorio entre 1 y 6, simulando la tirada del dado y en el segundo se muestra el valor del dado.
3. El mismo método jugar() también llama al método retornar_valor() de cada objeto Dado que devolverá el valor de cada uno de ellos. Ese valor devuelto se compara para determinar si los 3 dados son iguales (ganó) o no (perdió) dentro de una estructura condicional.

Acotación

Para cortar una línea en varias líneas en Python podemos encerrar entre paréntesis la condición:

```
if (self.dado1.retornar_valor()==self.dado2.retornar_valor()  
    and self.dado1.retornar_valor()==self.dado3.retornar_val  
or()):
```

O agregar una barra al final:

```
if self.dado1.retornar_valor()==self.dado2.retornar_valor()  
and \  
    self.dado1.retornar_valor()==self.dado3.retornar_valor()  
:
```

Variables de clase

Hemos visto cómo definimos atributos en una clase anteponiendo la palabra clave self:

```
class Persona:
    def __init__(self, nombre):
        self.nombre=nombre
```

Los atributos son independientes por cada objeto o instancia de la clase, es decir si definimos tres objetos de la clase Persona, todas las personas tienen un atributo nombre pero cada uno tiene un valor independiente.

En algunas situaciones necesitamos almacenar datos que sean compartidos por todos los objetos de dicha clase, en esas situaciones debemos emplear variables de clase.

Para definir una variable de clase lo hacemos dentro de la clase pero fuera de sus métodos:

```
class Persona:
    variable=20

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

Variables de clase

Problema 9:

Definir una clase Cliente que almacene un código de cliente y un nombre.

En la clase Cliente definir una variable de clase de tipo lista que almacene todos los clientes que tienen suspendidas sus cuentas corrientes.

Imprimir por pantalla todos los datos de clientes y el estado que se encuentra su cuenta corriente.

Cliente

atributos

código

nombre

métodos

__init__

imprimir

esta_suspendido

suspender

variables de clase

suspendidos (lista)

Se crearán 4 clientes:

1. Juan
2. Ana
3. Diego (cuenta suspendida)
4. Pedro (cuenta suspendida)

La clase Cliente define una variable de clase llamada suspendidos que es de tipo lista y por ser variable de clase es compartida por todos los objetos que definamos de dicha clase.

```
class Cliente:
    suspendidos=[] #Variable de Clase

    def __init__(self,codigo,nombre):
        self.codigo=codigo #Variable de Instancia
        self.nombre=nombre #Variable de Instancia
```

En el método imprimir mostramos el código, nombre del cliente y si se encuentra suspendida su cuenta corriente.

```
def imprimir(self):
    print("Codigo: {}".format(self.codigo))
    print("Nombre: {}".format(self.nombre))
    self.esta_suspendido()
```

El método suspender lo que hace es agregar el código de dicho cliente a la lista de clientes suspendidos.

```
def suspender(self):
    Cliente.suspendidos.append(self.codigo)
```

El método que analiza si está suspendido el cliente verifica si su código se encuentra almacenado en la variable de clase suspendidos.



```
def esta_suspendido(self):  
    if self.codigo in Cliente.suspendidos:  
        print("Esta suspendido")  
    else:  
        print("No esta suspendido")  
    print("_____")
```

Dentro del cuerpo principal del programa crearemos los 4 clientes (objetos)...

```
cliente1=Cliente(1,"Juan")  
cliente2=Cliente(2,"Ana")  
cliente3=Cliente(3,"Diego")  
cliente4=Cliente(4,"Pedro")
```

... y luego suspenderemos al cliente 3 y 4:

```
cliente3.suspender()  
cliente4.suspender()
```

Imprimiremos los datos de los 4 clientes:

```
cliente1.imprimir()  
cliente2.imprimir()  
cliente3.imprimir()  
cliente4.imprimir()
```

```
Codigo: 1  
Nombre: Juan  
No esta suspendido
```

```
Codigo: 2  
Nombre: Ana  
No esta suspendido
```

```
Codigo: 3  
Nombre: Diego  
Esta suspendido
```

```
Codigo: 4  
Nombre: Pedro  
Esta suspendido
```

terminal

*Es importante remarcar que todos los objetos acceden a una única lista llamada **suspendidos** gracias a que se definió como **variable de clase**.*



*Ejercicio_9_POO.p
y*

Podemos imprimir la variable de clase suspendidos de la clase Cliente:

```
print(Cliente.suspendidos)
```

```
[3, 4] terminal
```


Objetos dentro de objetos

Al ser las clases un nuevo tipo de dato se pueden poner en colecciones e incluso utilizarse dentro de otras clases.

```
class Pelicula:

    # Constructor de clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print('Se ha creado la película:', self.titulo)

    def __str__(self):
        return '{} ({}).format(self.titulo, self.lanzamiento)
```

continúa....

Objetos dentro de objetos

```
class Catalogo:

    peliculas = [] # Esta lista contendrá objetos de la clase Pelicula

    def __init__(self, peliculas=[]):
        Catalogo.peliculas = peliculas

    def agregar(self, p): # p será un objeto Pelicula
        Catalogo.peliculas.append(p)

    def mostrar(self):
        for p in Catalogo.peliculas:
            print(p) # Print toma por defecto str(p)
```

continúa....

Objetos dentro de objetos

```
#Programa principal
p = Pelicula("El Padrino", 175, 1972)
c = Catalogo([p]) # Añado una lista con una película desde el principio
c.mostrar()
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974)) # Añadimos otra
c.mostrar()
```

terminal

```
Se ha creado la película: El Padrino
El Padrino (1972)
Se ha creado la película: El Padrino: Parte
2
El Padrino (1972)
El Padrino: Parte 2 (1974)
```



*Ejercicio_12_POO.
py*

Encapsulación

El **encapsulamiento o encapsulación** 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 por defecto no oculta los atributos y métodos de una clase al exterior, por ejemplo:

```
class Clase:
    atributo_clase = "Hola"
    def __init__(self, atributo_instancia):
        self.atributo_instancia = atributo_instancia

mi_clase = Clase("Que tal")
print(mi_clase.atributo_clase)
print(mi_clase.atributo_instancia)

# 'Hola'
# 'Que tal'
```

Ambos atributos son perfectamente accesibles desde el exterior. Sin embargo esto es algo que tal vez no queramos. Hay ciertos métodos o atributos que queremos que pertenezcan **sólo a la clase o al objeto**, y que sólo puedan ser accedidos por los mismos. Para ello podemos usar la doble `__` para nombrar a un atributo o método. Esto hará que Python los interprete como *"privados"*, de manera que no podrán ser accedidos desde el exterior.

```
class Clase:
    atributo_clase = "Hola" # Accesible desde el exterior
    __atributo_clase = "Hola" # No accesible

    # No accesible desde el exterior
    def __mi_metodo(self):
        print("Haz algo")
        self.__variable = 0

    # Accesible desde el exterior
    def metodo_normal(self):
        # El método si es accesible desde el interior
        self.__mi_metodo()

mi_clase = Clase()
# mi_clase.__atributo_clase # Error! El atributo no es accesible
# mi_clase.__mi_metodo() # Error! El método no es accesible
print(mi_clase.atributo_clase) # Ok!
mi_clase.metodo_normal() # Ok!
```

Y como curiosidad, podemos hacer uso de **dir** para ver el listado de métodos y atributos de nuestra clase. Podemos ver claramente como tenemos el metodo_normal y el atributo de clase, pero no podemos encontrar __mi_metodo ni __atributo_clase.

```
print(dir(mi_clase))

#[ '_Class__atributo_clase', '_Class__mi_metodo', '_Class__variable',
# '_class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
# '_format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
# '_init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
# '_reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
# '_str__', '__subclasshook__', '__weakref__', 'atributo_clase', 'metodo_normal']
```

Pues bien, en realidad si que podríamos acceder a **__atributo_clase** y a **__mi_metodo** haciendo un poco de trampa. Aunque no se vea a simple vista, si que están pero con un nombre distinto, para de alguna manera ocultarlos y evitar su uso. Pero podemos llamarlos de la siguiente manera, pero por lo general **no es una buena idea**.

```
print(mi_clase._Class__atributo_clase)
# 'Hola'
mi_clase._Class__mi_metodo()
# 'Haz algo'
```

Fuente del ejemplo:

<https://ellibrodepython.com/encapsulamiento-poo>



encapsulamiento.p
y

Encapsulación: atributos privados

La **encapsulación** consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior, para **protegerlos**. En Python no existe, pero se puede simular precediendo atributos y métodos con **dos barras bajas** `__` como indicando que son “especiales”. En el caso de los atributos quedarían así:

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fue
ra."

e = Ejemplo()
print(e.__atributo_privado)
```

Y en los métodos...

```
class Ejemplo:
    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

e = Ejemplo()
e.__metodo_privado()
```

Encapsulación: atributos privados

¿Qué sentido tiene esto en Python? Ninguno, porque se pierde toda la gracia de lo que en esencia es el lenguaje: **flexibilidad** y **polimorfismo** sin control (veremos esto más adelante).

Sea como sea, para acceder a esos datos se deberían crear métodos públicos que hagan de interfaz. En otros lenguajes les llamaríamos **getters y setters** y es lo que da lugar a las *propiedades*, que no son más que atributos protegidos con interfaces de acceso.

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()
```

```
e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```

```
Soy un atributo inalcanzable desde fuera.
Soy un método inalcanzable desde fuera.
```

terminal

Getters y Setters en Python

- Los **getters** serían las funciones que nos permiten acceder a una variable privada. En Python se declaran creando una función con el decorador **@property**.
- Los **setters** serían las funciones que usamos para sobrescribir la información de una variable y se generan definiendo un método con el nombre de la variable sin guiones y utilizando como decorador el nombre de la variable sin guiones más ".setter".

```
class ListadoBebidas:

    def __init__(self):
        self.__bebida = 'Naranja'
        self.__bebidas_validas = ['Naranja', 'Manzana']

    @property
    def bebida(self):
        return "La bebida oficial es: {}".format(self.__bebida)

    @bebida.setter
    def bebida(self, bebida):
        self.__bebida = bebida
```

Getters y Setters en Python

En este ejemplo declaramos dos variables, una llamada `_bebida` y una lista llamada `_bebidas_validas`. Para recuperar la información de la variable `_bebida` tendremos que hacerlo con el objeto y el nombre de la función `bebida`.

```
#Programa principal
bebidas= ListadoBebidas()
print(bebidas.bebida)
bebidas.bebida = 'Limonada'
print(bebidas.bebida)
```



*Ejercicio_13_POO.
py*

```
La bebida oficial es: Naranja
La bebida oficial es: Limonada
```

terminal

Para ampliar (ejemplo):

https://pythones.net/propiedades-en-python-oop/#Propiedades_de_atributos_de_clase_en_Python_Getter_Setter_y_Deleter