Codo a Codo 4.0

FULL STACK PYTHON

html - css3 - bootstrap - javascript
vue.js - sql - python - django

Clase 30

Python – Parte 6

Temas que veremos las próximas clases

POO

- *Objetos*
- *Clases*
- *Abstracción*
- *Encapsulamiento*
- *Polimorfismo*
- *Herencia*

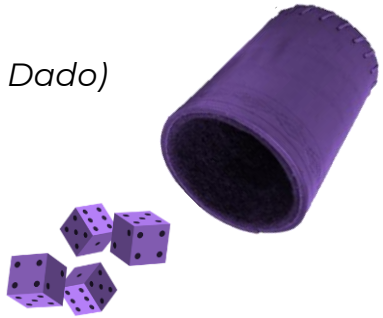
Colaboración de clases

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 JuegoDados, luego debemos definir los atributos y los métodos de cada clase:

Dado
- **atributos**
  - *valor*
- **métodos**
  - *__init__*
  - *tirar*
  - *imprimir*
  - *obtener_valor*

JuegoDados
- **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 __init__(self):
        self.valor = random.randint(1, 6)

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

    def imprimir_valor(self):
        print(f"Valor del dado: {self.valor}")

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

La clase JuegoDados define tres atributos de instancia del tipo Dado, en el método __init__ crea dichos objetos:

```
class JuegoDados:

    def __init__(self):
        self.dado_1=Dado()
        self.dado_2=Dado()
        self.dado_3=Dado()

    def jugar(self):
        self.dado_1.tirar()
        self.dado_1.imprimir()
        self.dado_2.tirar()
        self.dado_2.imprimir()
        self.dado_3.tirar()
        self.dado_3.imprimir()
        if self.dado_1.obtener_valor()==self.dado_2.obtener_valor()==\
            self.dado_3.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=JuegoDados()
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
```

Juego_dados.py

¿Cómo es, entonces, el flujo del programa?

1. Se crea el objeto de tipo JuegoDados que tendrá 3 dados (objetos).
2. Al llamar al método jugar() de la clase JuegoDados 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 obtener_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.

Atributos de clase

Hemos visto cómo definimos atributos de instancia anteponiendo la palabra clave self:

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

Los atributos de instancia 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
```

```python
# Bloque principal
persona1=Persona("Juan")
persona2=Persona("Ana")
persona3=Persona("Luis")

print(persona1.nombre) # Juan
print(persona2.nombre) # Ana
print(persona3.nombre) # Luis

print(persona1.variable) # 20
Persona.variable=5
print(persona2.variable) # 5
```

terminal

```
Juan
Ana
Luis
20
5
```

Se reserva solo un espacio para la variable "variable", independientemente que se definan muchos objetos de la clase Persona. La variable "variable" es compartida por todos los objetos persona1, persona2 y persona3.

Para modificar la variable de clase hacemos referencia al nombre de la clase y seguidamente el nombre de la variable:

```python
Persona.variable=5
```

Variables_de_clase.py

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

Encapsulamiento

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:

```python
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))

#['_Clase__atributo_clase', '_Clase__mi_metodo', '_Clase__variable',
#'__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
#'__format__', '__ge__', '__getattribute__', '__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._Clase__atributo_clase)
# 'Hola'
mi_clase._Clase__mi_metodo()
# 'Haz algo'
```

Fuente del ejemplo:
https://ellibrodepython.com/encapsulamiento-poo

encapsulamiento.py

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 fuera."

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? No mucho, porque se pierde toda la gracia de lo que en esencia busca el lenguaje: **flexibilidad** y **accesibilidad** 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()
```

terminal

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

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 sobreescribir 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

terminal

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

Para ampliar (ejemplo): https://pythones.net/propiedades-en-python-oop/#Propiedades_de_atributos_de_clase_en_Python_Getter_Setter_y_Deleter