Baldomero Sánchez

4. Objetos

Unidad 5

Programación Orientada a Objetos

Existen otros conceptos muy importantes y relacionados con la programación orientada a objetos. Aquí te los dejamos:

- Herencia
- Cohesión
- Abstracción
- Polimorfismo
- Acoplamiento
- Encapsulamiento

Métodos en Python: instancia, clase y estáticos

Tal como su nombre lo indica es una operación nula, o sea que no pasa nada cuando se ejecuta.
Se utiliza cuando se requiere por sintaxis una declaración pero no se quiere ejecutar ningún comando o código.
También se utiliza en lugares donde donde el código irá finalmente, pero no ha sido escrita todavía (utilizándolo como un relleno temporal, hasta que se escriba el código final)

- En un bucle no realiza ninguna iteración
- En una función esta sin definir que hace y retorno

```
# Primer ejemplo
for letra in "Python":
    if letra == "h":
        pass
    print ("Letra actual :" + letra)

# Segundo ejemplo
var = 10
while var > 0:
    var = var -1
    if var == 5:
        pass
    print ("Valor actual de la variable :" + str(var))

print ("fin del script")
```

Métodos en Python: instancia, clase y estáticos

En otros posts hemos visto como se pueden crear métodos con def dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia.
Haciendo uso de los decoradores, es posible crear diferentes tipos de métodos:

- Lo métodos de instancia **“normales”** que ya hemos visto como metodo()
- Métodos de clase **usando el decorador** @classmethod
- Y **métodos estáticos** usando el decorador @staticmethod

```
class Clase:

def metodo(self):
        return 'Método normal', self
@classmethod
def metododeclase(cls):
        return 'Método de clase', cls
@staticmethod
def metodoestatico():
        return "Método estático"
```

Métodos de instancia normal

Los métodos de instancia son los métodos normales, de toda la vida, que hemos visto anteriormente. Reciben como parámetro de entrada self que hace referencia a la instancia que llama al método. También pueden recibir otros argumentos como entrada.

- El uso de "**self**" es totalmente arbitrario.
- Se trata de una convención acordada por los usuarios de Python, usada para referirse a la instancia que llama al método, pero podría ser cualquier otro nombre.
- Lo mismo ocurre con "**cls**", que veremos a continuación.

```
class Clase:
def metodo(self, arg1, arg2):
        return 'Método normal', self



mi_clase = Clase()
mi_clase.metodo("a", "b")
# ('Método normal', <__main__.Clase at 0x10b9daa90>)
```

Métodos de clase (classmethod)

A diferencia de **los métodos de instancia**, los métodos de clase reciben como argumento cls, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia.

```
class Clase:
        @classmethod
        def metododeclase(cls):
                return 'Método de clase', cls
```

Se pueden llamar sobre la clase.

```
Clase.metododeclase()
# ('Método de clase', __main__.Clase)
```

Pero también se pueden llamar sobre el objeto.

Los métodos de clase:

- No pueden acceder a los atributos de la instancia.
- Pero si pueden modificar los atributos de la clase

```
mi_clase=Clase()
mi_clase.metododeclase()
# ('Método de clase', __main__.Clase)
```

Métodos estáticos (staticmethod)

Los métodos estáticos se pueden definir con el decorador @staticmethod y **no aceptan como parámetro ni la instancia ni la clase**.
Es por ello por lo que no pueden modificar el estado ni de la clase ni de la instancia. Pero por supuesto pueden aceptar parámetros de entrada.

Los métodos estáticos se podrían ver como funciones normales, con la salvedad de que van ligadas a una clase concreta

```
class Clase:

        @staticmethod
        def metodoestatico():
                return "Método estático"
```

```
mi_clase = Clase()
Clase.metodoestatico()
mi_clase.metodoestatico()
```

```
mi_clase.metododeclase()
# ('Método de clase', __main__.Clase)
```

Staticmethod vs staticmethod

Explicación

Map(fun, iter)
map () devuelve un objeto de mapa (que es un iterador) de los resultados después de aplicar la función dada a cada elemento de un iterable dado (lista, tupla, etc.)

```
class Date(object):

    def __init__(self, day=0, month=0, year=0):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999

date2 = Date.from_string('11-09-2012')
is_date = Date.is_date_valid('11-09-2012')
```

```
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

```
text = 'Python'
# convert string to list
text_list = list(text)
print(text_list)
# check type of text_list
print(type(text_list))
# Output: ['P', 'y', 't', 'h', 'o', 'n']
 # <class 'list'>
```

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. Además de ello, una clase hija puede sobreescribir los métodos o atributos, o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos como se puede usar la herencia en Python, con la clase Perro que hereda de Animal

La clase Perro es la hija de Animal usando __bases__

```
# Definimos una clase padre
class Animal:
        pass
# Creamos una clase hija que hereda de la padre
class Perro(Animal):
        pass
```

```
print(Perro.__bases__)
# (<class '__main__.Animal'>,)
```

```
###
        De manera similar podemos ver que clases descienden de una en concreto con __subclasses__.
###

print(Animal.__subclasses__())
# [<class '__main__.Perro'>]
```

Herencia – Extendiendo y modificando métodos

Vamos a definir una clase padre Animal que tendrá todos los atributos y métodos genéricos que los animales pueden tener. Esta tarea de buscar el denominador común es muy importante en programación. Veamos los atributos: especie y edad

Tenemos la especie ya que todos los animales pertenecen a una.
Y la edad, ya que todo ser vivo nace, crece, se reproduce y muere.
Y los métodos o funcionalidades:

- Tendremos el método **hablar,** que cada animal implementará de una forma. Los perros ladran, las abejas zumban y los caballos relinchan.
- Un método **moverse**. Unos animales lo harán caminando, otros volando.

Y por último un método descríbeme que será común.

```
###
Definimos la clase padre, con una serie de atributos
comunes para todos los animales
###
class Animal:
    def __init__(self, especie, edad):
            self.especie = especie
            self.edad = edad
    # Método genérico pero con implementación particular
    def hablar(self):
     # Método vacío
            pass
    # Método genérico pero con implementación particular
    def moverse(self):
    # Método vacío
            pass
    # Método genérico con la misma implementación
    def describeme(self):
            print("Soy un Animal del tipo", type(self).__name__)
```

Herencia – Extendiendo y modificando métodos (2)

Vamos a crear varios animales concretos y sobreescrbir algunos de los métodos que habían sido definidos en la clase Animal, como el hablar o el moverse, ya que cada animal se comporta de una manera distinta.

Podemos incluso crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la Abeja con picar().

```
class Perro(Animal):
        def hablar(self):
                print("Guau!")
        def moverse(self):
                print("Caminando con 4 patas")
class Vaca(Animal):
        def hablar(self):
                print("Muuu!")
        def moverse(self):
                print("Caminando con 4 patas")
class Abeja(Animal):
        def hablar(self):
                print("Bzzzz!")
        def moverse(self):
                print("Volando")
        # Nuevo método
        def picar(self):
                print("Picar!")
```

Herencia – Extendiendo y modificando métodos (3)

Podemos crear nuestros objetos de esos animales y hacer uso de sus métodos que podrían clasificarse en tres:

- Heredados directamente de la clase padre: **describeme()**
- Heredados de la clase padre pero modificados: **hablar()** y **moverse()**
- Creados en la clase hija por lo tanto no existentes en la clase padre: **picar()**

```
mi_perro = Perro('mamífero', 10)
mi_vaca = Vaca('mamífero', 23)
mi_abeja = Abeja('insecto', 1)

mi_perro.hablar()
mi_vaca.hablar()
# Guau!
# Muuu!

mi_vaca.describeme()
mi_abeja.describeme()
# Soy un Animal del tipo Vaca
# Soy un Animal del tipo Abeja

mi_abeja.picar()
# Picar!
```

Herencia – Uso de super()

La función **super()** nos permite acceder a los métodos de la clase padre desde una de sus hijas. Volvamos al ejemplo de Animal y Perro.

Queramos que nuestro Perro tenga un parámetro extra en el constructor, como podría ser el dueño. Para realizar esto tenemos dos alternativas:

Podemos crear un nuevo __init__ y guardar todas las variables una a una.
O podemos usar super() para llamar al __init__ de la clase padre que ya aceptaba la especie y edad, y sólo asignar la variable nueva

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    def hablar(self):
        pass
    def moverse(self):
        pass
    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

```
class Perro(Animal):
    def __init__(self, especie, edad, dueño):
        # Alternativa 1
        # self.especie = especie
        # self.edad = edad
        # self.dueño = dueño
        # Alternativa 2
        super().__init__(especie, edad) #invoca Animal __init__
        self.dueño = dueño
```

```
mi_perro = Perro('mamífero', 7, 'Luis')
mi_perro.especie
mi_perro.edad
mi_perro.dueño
```

Herencia – múltiple

En Python es posible realizar herencia múltiple.
La herencia múltiple es similar, pero una clase hereda de varias clases padre en vez de una sola.

Veamos un ejemplo:

- Por un lado tenemos dos clases Clase1 y Clase2, y por otro tenemos la Clase3 que hereda de las dos anteriores.
- Por lo tanto, heredará todos los métodos y atributos de ambas.

```
class Clase1:
        pass
class Clase2:
        pass
class Clase3(Clase1, Clase2):
        pass
```

```
# Es posible también heredar de otra clase

class Clase1:
        pass
class Clase2(Clase1):
        pass
class Clase3(Clase2):
        pass
```

Herencia – múltiple (2)

Si llamo a un método que todas las clases tienen en común ¿a cuál se llama?. Pues bien, existe una forma de saberlo.

La forma de saber a que método se llama es consultar el **MRO o *Method Order Resolution***.

- Esta función **nos devuelve una tupla** con el orden de búsqueda de los métodos.
- Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

```
class Clase1:
        pass
class Clase2:
        pass
class Clase3(Clase1, Clase2):
        pass
        print(Clase3.__mro__)
# Resultado de la consulta MRO
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class '__main__.Clase2'>, <class 'object'>
```

Herencia – múltiple (3)

Una curiosidad es que al final del todo vemos la clase object. Aunque pueda parecer raro, es correcto ya que en realidad todas las clases en Python heredan de una clase genérica object, aunque no lo especifiquemos explícitamente.

Y como último ejemplo,…el cielo es el límite. Podemos tener una clase heredando de otras tres. Fíjate en que el MRO depende del orden en el que las clases son pasadas: 1, 3, 2.

```
class Clase1:
        pass
class Clase2:
        pass
class Clase3:
        pass
class Clase4(Clase1, Clase3, Clase2):
        pass print(Clase4.__mro__)
# (<class '__main__.Clase4'>, <class '__main__.Clase1'>, <class '__main__.Clase3'>, <class '__main__.Clase2'>, <class 'object'>)
```

Cohesión en Programación

La cohesión hace referencia al grado de relación entre los elementos de un módulo.
En el diseño de una función, es importante pensar bien la tarea que va a realizar, intentando que sea única y bien definida.
Cuantas más cosas diferentes haga una función sin relación entre sí, más complicado será el código de entender.
Existen por lo tanto dos tipos de cohesión:

- **La cohesión débil** que indica que la relación entre los elementos es baja. Es decir, no pertenecen a una única funcionalidad.
- **La cohesión fuerte**, que debe ser nuestro objetivo al diseñar programas. La cohesión fuerte indica que existe una alta relación entre los elementos existentes dentro del módulo.

Normalmente acoplamiento débil se relaciona con cohesión fuerte o alta.

Cohexión Débil

Tenemos una función suma() que suma dos números.

El problema es que además de sumar dos números, los convierte a float() y además pide al usuario que introduzca por pantalla el número.

Podría parecer que esas otras dos funcionalidades no son para tanto, pero si por ejemplo una persona quiere usar nuestra función suma() pero ya tiene los números y no quiere pedirlos por pantalla, no le serviría nuestra función.

```
def suma():
        num1 = float(input("Dame primer número"))
        num2 = float(input("Dame segundo número"))
        suma = num1 + num2
        print(suma)

suma()
```

Cohexión fuerte

La función tuviese una cohesión fuerte, sería conveniente que la suma realizara una única tarea bien definida, que es sumar.

```
# Bien. Cohesión fuerte

def suma(numeros):
        total = 0
        for i in numeros:
                total = total + i
        return total

Suma(3,2,23,7,6)
```

Abstracción

La abstracción es un termino que hace referencia a la **ocultación de la complejidad intrínseca** de una aplicación al exterior, centrándose sólo en como puede ser usada, lo que se conoce como **interfaz**.

Al hablar del enfoque **caja negra**, es un concepto muy relacionado.
Dicho en otras palabras, la abstracción consiste en ocultar toda la complejidad que algo puede tener por dentro, ofreciéndonos unas funciones de alto nivel, por lo general sencillas de usar, que pueden ser usadas para interactuar con la aplicación sin tener conocimiento de lo que hay dentro.

NOTA: Es posible crear métodos abstractos en Python con decoradores como **@absttractmethod**

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.

¿Pero qué significa esto?

Pues bien, significa que objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.

En lenguajes de programación como Python, **que tiene tipado dinámico**, el polimorfismo va muy relacionado con el **duck typing.**

Duck typing o **tipado pato** el estilo de tipificación dinámica de datos en que *el conjunto actual de métodos y propiedades* determina la validez semántica, en vez de que lo hagan la herencia de una clase en particular o la implementación de una interfaz específica

Polimorfismo en Python

```
class Animal:
        def hablar(self):
                pass
```

```
# Otras dos clases que heredan de la anterior
class Perro(Animal):
        def hablar(self):
                print("Guau!")
class Gato(Animal):
        def hablar(self):
                print("Miau!")
```

```
###
 A continuación creamos un objeto de cada clase
y llamamos al método hablar(). Podemos
observar que cada animal se comporta de manera
distinta al usar hablar().
###

for animal in Perro(), Gato()
                animal.hablar()
# Guau!
# Miau!
```