



Curso Intensivo de Python - Tema 8

Agustín Arenas



Curso Intensivo de Python - Tema 8

- 1) Objetos
- 2) Clases
- 3) Documentación
- 4) Clases y Objetos II
- 5) Clases y Objetos III
- 6) F-strings



Objetos

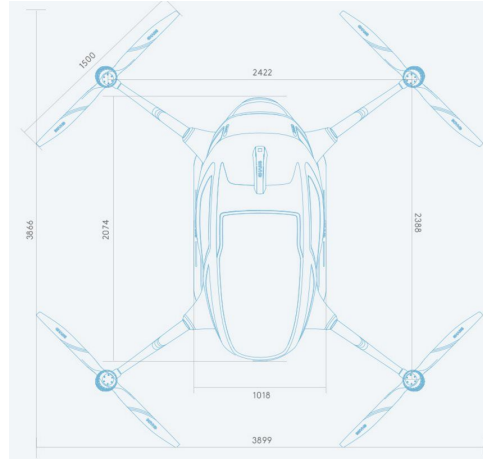
- Hasta ahora hemos llevado un estilo de programación conocido como **procedural**, hemos enfocado nuestro esfuerzo en describir los procedimientos o acciones que se llevan a cabo
- Sin embargo, existen otros estilos como la **programación funcional, orientado a datos, orientado a objetos, etc.** Este último va a ser nuestro pequeño caso de estudio
- El paradigma orientado a objetos se basa en 4 pilares:
 - **Abstracción:** Nos abstraemos del funcionamiento interno del objeto y tomamos un modelo de caja negra, es decir, nos enfocamos en las entradas y salidas
 - **Encapsulación:** Se basa en la idea de esconder los datos y métodos en una sola unidad, la clase. Es decir, ningún objeto debería conocer toda la información de otro objeto
 - **Herencia:** En resumidas palabras, podemos crear clases a partir de una clase padre
 - **Polimorfismo:** Aquí un objeto es polimórfico




Objetos

- Dado el alcance de este curso, sólo veremos la aplicación de abstracción y encapsulación
- Un objeto podemos decir que es una colección de datos con un comportamiento asociado
- Pero a la hora de crear un objeto, necesitamos de un **plano**

Clases



- Este plano se denomina de forma técnica, **clase**. ¿Cuál es la diferencia entre objeto y clase?
- Las clases se usan para describir los objetos (Es por esto la idea de plano) y cada instancia de la clase es un objeto



```
import random as rd
```

```
class Moneda:
```

```
    #Para crear una clase se usa la palabra
```

```
    #reservada class seguida del nombre de la clase.
```

```
    def __init__(self):
```

```
        #Este método es una característica de Python que se usa
```

```
        #para dos razones:
```

```
        #Primero inicializa el objeto, poniendolo en un estado apropiado
```

```
        #cuando se crea
```

```
        #Segundo es que __init__ puede tomar muchas formas, permitiendo
```

```
        #poder definir como se crea el objeto o inicializa
```

```
        self.cara_cruz = 'cara'
```

```
    def lanzar(self):
```

```
        '''
```

```
        Método que genera un número aleatorio y determina el estado de la moneda
```

```
        @param Ninguno
```

```
        @return Nada
```

```
        '''
```

```
        #self es lo que marca la diferencia entre método o función
```

```
        # (Si bien son prácticamente lo mismo).
```

```
        #Es un análogo al "this" de otros lenguajes,
```

```
        # y lo que hace es una referencia al objeto que
```

```
        # el método hace cuando se invoca.
```

```
        #Podemos acceder a atributos y métodos
```

```
        # de un objeto como si fuera cualquier otro objeto.
```

```
        if (rd.randint(0,1)):
```

```
            self.cara_cruz = 'cara'
```

```
            print('Toss a coin to your Witcher...')
```

```
        else:
```

```
            self.cara_cruz = 'cruz'
```

```
            print('O\ Valley of Plenty...')
```

Clases y Objetos

Clases y Objetos

```
def getEstado(self):
    """
    Devuelve el estado de la moneda
    @param Ninguno
    @return Estado de la moneda 'cara' o 'cruz'
    """
    return self.cara_cruz

def setEstado(self,estado):
    """
    Establece el estado de la moneda
    @param Nuevo estado
    @return Nada
    """
    self.cara_cruz = estado

#Los métodos get y set se llaman mutadores y son la principal herramienta
#para poder establecer (set) u obtener (get) el estado de un objeto
def main():
    moneda_geralt = Moneda() #Con esta linea, creamos una nueva instancia de
    #un objeto moneda
    print(moneda_geralt.getEstado()) #Invocamos el metodo get
    for l in range(3):
        moneda_geralt.lanzar()
        print(moneda_geralt.getEstado())
    if moneda_geralt.getEstado() == 'cara':
        moneda_geralt.setEstado('cruz')
    else:
        moneda_geralt.cara_cruz = 'cara'
        print(moneda_geralt.cara_cruz )
    print('Fin programa')

main()
```



Documentación

- Los comentarios entre `'''` (o `"""`) se escriben después del método y constituye la documentación de la clase. Es una práctica a implementar siempre que podamos
- `@param` indica los parámetros del método y `@return` lo que este devuelve. Podemos leer esto haciendo en consola **`python -i archivo.py`** y luego **`help(NombreClase)`**



Clases y Objetos II

- Lo otro es que hemos roto la encapsulación
- En la línea 58 hemos accedido directamente a la variable y evitamos el mutador set, lo cual está mal porque el objeto tiene sus datos de forma pública.
- Para remendar esto, debemos hacer que la variable **cara_cruz** sea **privada**. Para python, todos los métodos y variables son **públicos** por defecto



Clases y Objetos II

- Si deseamos mantener la encapsulación, debemos añadir un prefijo de dos `__` lo cual lleva a cabo un **name mangling** sobre el atributo
- Esto significa que el atributo es "privado"
- **¡Ojo!** Las reglas de name mangling están diseñadas principalmente para evitar accidentes; todavía es posible acceder o modificar una variable que se considera privada. Esto incluso puede resultar útil en circunstancias especiales, como en el debugger



Clases y Objetos II

```
def main():
    moneda_geralt = Moneda()
    print(moneda_geralt.getEstado())
    for l in range(3):
        moneda_geralt.lanzar()
        print(moneda_geralt.getEstado())
    if moneda_geralt.getEstado() == 'cara':
        moneda_geralt.setEstado('cruz')
    else:
        moneda_geralt.__cara_cruz = 'cara'
    print('Estado final',moneda_geralt.getEstado())
    print('Fin programa')
main()
```

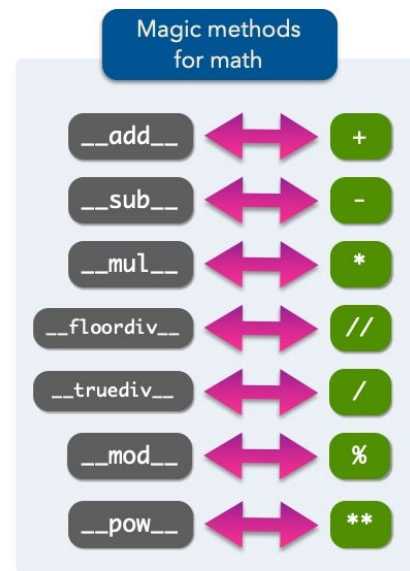
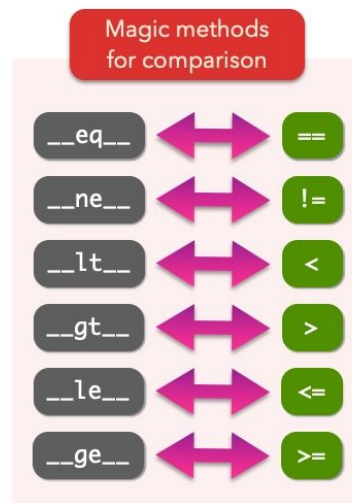


Clases y Objetos III

- Podemos pensar en la clase como el “**molde**” con el que se crean nuevos objetos de ese tipo
- El constructor (`__init__()`) permite crear objetos cambiando los atributos que deseemos cambiar
- Para esto se define la función para que reciba argumentos **def** `__init__(self, argumentos)`

Clases y Objetos III

- Existen métodos que se utilizan para comparar, hacer operaciones matemáticas o solamente explicar los objetos
- `__str__` y permite establecer la forma en la que un objeto es representado como cadena de texto





```
class Animales:
    def __init__(self, nombre, cantidad_patas, sonido):
        self.nombre = nombre
        self.cantidad_patas = cantidad_patas
        self.sonido = sonido
    def getNombre(self):
        """
        Devuelve el nombre del animal
        @param Ninguno
        @return nombre del animal
        """
        return self.nombre
    def getCantidadPatras(self):
        """
        Devuelve la cantidad de patas del animal
        @param Ninguno
        @return cantidad de patas del animal
        """
        return self.cantidad_patas
    def getSonido(self):
        """
        Devuelve el sonido del animal
        @param Ninguno
        @return sonido del animal
        """
        return self.sonido
    def setNombre(self, nombre):
        """
        Establece el nombre del animal
        @param Nuevo nombre
        @return Nada
        """
        self.nombre = nombre
```

Clases y Objetos III

Clases y Objetos III

```
def setCantidadPatas(self, cantidad_patas):
    """
    Establece la cantidad de patas del animal
    @param Nueva cantidad de patas
    @return Nada
    """
    self.cantidad_patas = cantidad_patas
def setSonido(self, sonido):
    """
    Establece el sonido del animal
    @param Nuevo sonido
    @return Nada
    """
    self.sonido = sonido
def __str__(self):
    """
    Devuelve una cadena con los datos del animal
    @param Ninguno
    @return Cadena con los datos del animal
    """
    return f'Nombre: {self.nombre} - Cantidad de patas: {self.cantidad_patas} - Sonido: {self.sonido}' #
f-strings: formateo de cadenas desde Python 3.6
def __eq__(self, otro_animal):
    """
    Compara dos animales
    @param Otro animal
    @return True si son iguales, False si son diferentes
    """
    return (self.nombre == otro_animal.nombre
            and self.cantidad_patas == otro_animal.cantidad_patas
            and self.sonido == otro_animal.sonido)
```

Clases y Objetos III

```
def main():
    animal_1 = Animales('Perro', 4, 'Guau')
    animal_2 = Animales('Gato', 4, 'Miau')
    animal_3 = Animales('Pájaro', 2, 'Pio')
    animal_4 = Animales('Vaca', 4, 'Muu')

    print(animal_1)
    print(animal_2)
    print(animal_3)
    print(animal_4)

    print(animal_1 == animal_2) # Llamada implícita a __eq__
    print(animal_1.__eq__(animal_4))
main()
```


F-strings

Table of Contents

7. Input and Output

- 7.1. Fancier Output
 - 7.1.1. Formatted String Literals
 - 7.1.2. The String format() Method
 - 7.1.3. Manual String Formatting
 - 7.1.4. Old string formatting
- 7.2. Reading and Writing Files
 - 7.2.1. Methods of File Objects
 - 7.2.2. Saving structured data with json

Previous topic

6. Modules

Next topic

8. Errors and Exceptions

This Page

Report a Bug
Show Source

7.1.1. Formatted String Literals

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Passing an integer after the `:'` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd      ==>      4127
Jack        ==>      4098
Dcab        ==>      7678
```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies `ascii()`, `'!s'` applies `str()`, and `'!r'` applies `repr()`.

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the [reference guide for the Format Specification Mini-Language](#).

7.1.2. The String format() Method

Link de documentación



¿Preguntas?