FSCheckPoint6

July 24, 2024

#¿Para qué usamos Clases en Python?

Una clase es una agrupación de código que sirve para crear prototipos de objetos. Estos objetos pueden tener variables y/o constantes, que denominamos *propiedades*; y funciones, que denominamos *métodos*. La clase nos define que propiedades y métodos tendrán los objetos de dicha clase. Una vez definida esta, para crear un objeto debemos declararlo en el código indicando que el objeto es de dicha clase, lo que se denomina instanciar y que, esencialmente, significa que se reserva la cantidad de memoria del ordenador necesaria para contener los métodos y propiedades de la clase. El código siguiente define la clase *toyClass*, la cual consta de una propiedad (*propOne*) y un método (*methodOne*) y crea el objeto *toyClassInstance* de clase toyClass. Como se puede apreciar, la sintaxis del proceso de instanciación es parecida a una llamada a una función.

[ ]:
```
class toyClass:
  propOne = 5
  def methodOne(self):
    return'I am a method of the toyClass'

toyClassInstance = toyClass()

print(toyClassInstance.propOne)
print(toyClassInstance.methodOne())
```

```
5
I am a method of the toyClass
```

Los métodos requieren al menos del argumento self, el cual hace referencia a la posición en memoria donde se almacena el objeto una vez instanciado.

#¿Qué es un método dunder?

Los métodos dunder son métodos especiales que no son invocados directamente por el programador sino que se invocan internamente cuando se realiza cierta acción. Estos métodos empiezan y terminan con dos guiones bajos (en inglés "underscore"), de donde reciben el nombre de dunder ("double underscore"). Para entender estos métodos debemos tener claro que todo en Python es un objeto. Así pues, por ejemplo, cuando sumamos dos números enteros, el programador usa el símbolo "+" mientras que internamente, se llama al método ___add___(), el cual define cómo se realiza dicha operación. Para visualizar esto podemos usar el código siguiente:

[ ]:
```
x = 2
y = 5
```

```
x + y
print(type(y))
print(type(x))
print (dir(type(y)))
print (dir(type(x)))
```

```
<class 'int'>
<class 'int'>
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio',
'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio',
'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

Como se puede apreciar en el código anterior, los objetos x e y son instancias de la clase "int" (entero) y, entre otros, tienen definido el método *___add___* que es llamado cuando usamos el simbolo +.

Técnicamente, este modo de proceder se denomina sobrecarga de operadores ("operators overload" en inglés) y, desde un punto de vista práctico, nos permite definir qué procedimiento debe ejecutarse cuando usamos un operador (por ejemplo, "+") con los miembros de las clases que creemos.

#¿Qué método se ejecuta automáticamente cuando se crea una instancia de una clase?

Como dijimos anteriormente, los métodos dunder se invocan en segundo plano al realizar una acción como, por ejemplo, la invocación del método ___add___() al utilizar el operador "+". Otro ejemplo relevante es el método *___init___()*, el cual se ejecuta en el momento en que creamos una instancia de una clase. El uso típico de este método es el de actuar como inicializador o, tal y

como se denomina en el paradigma de programación basado en objetos, de “constructor”.

Retomando la definición de la clase “toyClass” vista anteriormente, podemos desarrollar el ejemplo ilustrativo mostrado debajo. En este caso, en vez de asumir un valor para la propiedad de la clase (propOne), vamos a implementar un método ___init___() que nos permita asignar un valor que introduciremos como argumento al instanciar la classe, el valor 497 en el ejemplo.

[ ]:
```
class toyClass:

  def __init__(self,propOne):
    self.propOne = propOne

  def methodOne(self):
    return f'The value of propOne is {self.propOne}'

toyClassInstance = toyClass(497)

print(toyClassInstance.propOne)
print(toyClassInstance.methodOne())
```

```
497
The value of propOne is 497
```

#¿Qué es un decorador de python?

Un *decorador*, o función decoradora, es una función que toma otra función como argumento y que añade codigo creando una función diferente. La utilidad de los decoradores recae en que podemos modificar el comportamiento de una función sin tener que modificar su implementación. Veamos un ejemplo ilustrativo:

[ ]:
```
def decorator(func):
  def inner_wrapper():
    print("I am about to call the function I am decorating")
    func()
    print("I just ran the function and I keep adding content to it")
  return inner_wrapper

def cheese():
  print("Cheese!")

print(decorator(cheese)) # Imprimiendo la referencia a la función inner_wrapper

cheese = decorator(cheese) #Decorando la función cheese

cheese() #Corremos la función decorada
```

```
<function decorator.<locals>.inner_wrapper at 0x78dac8ab70a0>
I am about to call the function I am decorating
Cheese!
I just ran the function and I keep adding content to it
```

Tal y como se muestra en el ejemplo, se ha creado la función *decorator*, la cual requiere una función como argumento y tiene otra función enlazada en su interior, la función *inner_wrapper*. La función decorator devuelve la función modificada pero sin usar parenthesis. Esto significa que en vez de evaluar la función y devolver el resultado, devuelve una referencia a la misma función. Esto nos permite usar la función enlaçada inner_wrapper que solo podría ser invocada desde dentro de la función decorator.

Lo mostrado en el ejemplo anterior ilustra el uso de funciones decoradoras en Python. La sintaxis que hemos usado es completamente funcional pero existe una sintaxis alternativa mucho más elegante y legible. Esta se basa en la utilización del símbolo "@". Este símbolo, seguido del nombre de la función decoradora, se coloca justo antes de la definición de la función que será decorada. Una vez aplicado al ejemplo anterior obtenemos el siguiente código de programación, más corto pero equivalente al anterior:

[ ]:
```
def decorator(func):
  def inner_wrapper():
    print("I am about to call the function I am decorating")
    func()
    print("I just ran the function and I keep adding content to it")
  return inner_wrapper

@decorator   #Equivalente a cheese = decorator(cheese)
def cheese():
  print("Cheese!")

cheese() #Corremos la función decorada
```

```
I am about to call the function I am decorating
Cheese!
I just ran the function and I keep adding content to it
```

#¿Qué es el polimorfismo?

En Python, podemos distinguir tres tipos de polimorfismos, los denominados polimorfismo a través de funciones, polimorfismo de clases y polimorfismo de clases heredadas.

- Polimorfismo a través de funciones (o funcional). Este concepto hace referencia a que una misma función puede ser usada por objetos de tipo distinto, adaptando su comportamiento en función de este. Un ejemplo característico es la función length (len()), que cambia el comportamiento en función del tipo de objeto que pasamos como argumento. Así pues, si el argumento es una cadena de caracteres, la función len() nos devuelve el número de carácteres de la cadena pero si, por ejemplo, el argumento es un diccionario, retorna el número de pares clave/valor.
- Polimorfismo a través de clases. Distintas clases pueden implementar uno o varios métodos con el mismo nombre aunque con definiciones distintas para cada clase. Esto puede aprovecharse en el código para ejecutar el mismo método para las tres clases mediante, por ejemplo, un bucle for o una función. Por ejemplo, supongamos que tenemos las clases perro, pato y cerdo, todas ellas con un método denominado habla. Aprovechando que todas tienen el método habla, podemos invocarlo dentro del bucle for, sin importar que sean de clases distintas,

como sigue:

[ ]:
```
class perro:
  def habla(self):
    print("Bup, bup!")

class pato:
  def habla(self):
    print("Cua, cua!")


class cerdo:
  def habla(self):
    print("Oink, oink!")

xopi = perro()
donald = pato()
babe = cerdo()

for animal in (xopi, donald, babe):
  animal.habla()
```

```
Bup, bup!
Cua, cua!
Oink, oink!
```

- Polimorfismo de herencia de clase. Este tipo de polimorfismo hace referencia a que el método polimórfico puede incluso ser heredado de una clase padre sin ser implementado explícitamente. Retomando el ejemplo anterior, asumamos que existe una clase padre denominada animal la cual se usa para crear las clases perro, pato y cebra. Las dos primeras modifican el método habla de la clase padre pero la tercera no. Podemos comprobar que en este caso, aunque la clase cebra ha heredado el método habla de la clase padre animal que puede ser invocada dentro del bucle for, sin importar que no se haya definido explícitamente:

[ ]:
```
```

[ ]:
```
class animal:
  def habla(self):
    print("Hablo, hablo!")

class perro(animal):
  def habla(self):
    print("Bup, bup!")

class pato(animal):
  def habla(self):
    print("Cua, cua!")
```

```
class cebra(animal):
  pass

xopi = perro()
donald = pato()
cebri = cebra()

for ani in (xopi, donald, cebri):
  ani.habla()
```

```
Bup, bup!
Cua, cua!
Hablo, hablo!
```

#¿Qué es una API?

Una API, del inglés *Application programming interface*, o, en español, Interfaz de programación de aplicaciones, hace referencia a una pieza de código que permite la comunicación entre dos sistemas o aplicaciones. Si lo comparamos a la acción de ir a un restaurante a comer, el cliente desea hacer un pedido y la cocina es la encargada de prepararlo. De manera similar a la función de una API, el camarero es el que permite la conexión entre el cliente, que pide la comida, y la cocina. Primero el camarero crea una nota y la cocina responde a esa nota preparando los platos que, finalmente, son llevados al cliente por el camarero. Cada API es distinta y normalmente viene acompañada de instrucciones para aprender a usarla.

#¿Cuáles son los tres verbos de API?

Existen diferentes tipos de APIS con estándares de comunicación distintos. Todas ellas usan los denominados verbos API, que permiten la realización de distintas acciones. Aunque existen otros, los verbos más populares son:

- GET. Usado para solicitar datos o recursos.
- POST. Usado para enviar datos destinados a la creación de un recurso.
- PUT. Usado para actualizar un recurso.
- PATCH. Usado para actualizar una sección específica de un recurso.
- DELETE. Usado para eliminar por completo un recurso.

Algunas aplicaciones pueden centrarse en usar solo los verbos POST, PUT y PATCH.

#¿Qué es Postman?

Postman es una plataforma que te permite conectar con APIs externas o locales. Postman se encarga de la realización de las acciones dictadas por los verbos API y es capaź de formatear los datos devueltos por las APIs, por ejemplo, usando estándar JSON (JavaScript Object Notation), el cual permite una mayor legibilidad (y por tanto comprensión) de los datos.

Esta funcionalidad es de utilidad en el proceso de diseño, construcción y testeo de las APIs que se están desarrollando sin la necesidad de desplegar completamente la API en un servidor.

#¿Es MongoDB una base de datos SQL o NoSQL?

Existen dos grandes grupos de bases de datos, las *relacionales o SQL* (Structured Query Language) y las *no relacionales o NoSQL*. Las bases de datos SQL se desarrollaron los años 70 y organizan

la información en tablas, de manera similar a como funcionaria una hoja de cálculo. Cada entidad independiente (por ejemplo, likes, profesores, alumnos, etc) es almacenada en una tabla donde se relacionan registros y campos (p.ej, nombre, edad, dirección, etc). En términos generales este tipo de base de datos prioriza la integridad de la información a costa de una penalización en la velocidad.

Las bases de datos no relacionales o noSQL difieren en el sentido de que no tienen una estructura rígida y permiten la redundancia de datos, lo que permite una mayor velocidad de acceso a costa de un enfoque menos basado en la integridad de la información. Al no existir un estándar de referencia, cada motor puede seguir una estructura distinta. Estas bases de datos se pueden dividir en tres tipos principales: *bases de datos clave-valor*, que organizan la información en parejas de clave valor de manera parecida a un diccionario enciclopédico; las *bases de datos documentales* que guardan la información en archivos que toman como base de la información documentos codificados en algún formato estándar, como JSON; y las *bases de datos de grafos*, las cuales organizan la información en nodos que se van conectando.

MongoDB es una base de datos documental y por tanto NoSQL. En MongoDB se pueden crear colecciones de documentos, lo cual podría asemejarse a una tabla pero tiene la particularidad de que no existe un esquema de documento fijo y cada uno puede ser distinto. Es decir, contener más campos o de longitud distinta.