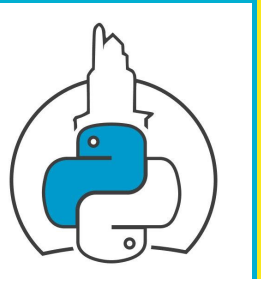
PYTHON CORUÑA - TECH TALK



Python Dataclass

Tech-talk

Dataclass - Definición

Definición en una frase:

Son clases que tienen implementadas por defecto las siguientes funcionalidades básicas:

- inicialización de instancias: __init__()
- comparación:__eq__()
- representación __repr__()

- Se diseñaron para modelar datos a través de atributos y métodos pero realmente NO aplican restricciones su uso

- Por tanto pueden ser útiles para sustituir a otras estructuras de datos

DataClass Conceptos básicos - Magic/Dunder/Special methods

- Son los métodos que definen cómo se comporta un objeto al utilizar los operadores nativos del lenguaje ó funciones built-in.
- La definición de estos métodos es lo que permite que, 2 métodos built-in ó 2 operadores iguales se comporten de forma distinta dependiendo del tipo de dato sobre el que se aplican

```
>>> # Adds the two numbers
>>> 1 + 2
3
>>> # Concatenates the two strings
>>> 'Hola' + 'Mundo'
'HolaMundo'
```

- Se definen con dos underscores: *__nombre__()*

```
>>> a: int = 5
>>> dir(a)
['__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__',
```

Clases Conceptos básicos - métodos *__new__( ) vs __init__( )*

- __new__( ) e __init__( ) son llamados cuando se instancia una clase, pero __new__( ) se ejecuta de forma implícita

- __new__( ) : **Crea una instancia de la clase**
  Se ejecuta cada vez que se instancia, debe devolver algo para que a continuación se ejecute __init__()
  __new__( ) es un método estático de la clase *object* con la firma:
  ```
  object.__new__(class, *args, **kwargs)
  ```

- __init__( ) : **Inicializa la instancia**
  Se ejecuta cuando __new__ () le pasa una instancia en self
  No debe devolver nada (TypeErr).

Clases Conceptos básicos - métodos *__new__()* vs *__init__()*

```python
class Person:
    def __init__(self, name):
        self.name = name

```

```python
person = Person("John")
print(person.__dict__)
```

```
{'name': 'John'}
```

*object.__new__(class, *args, **kwargs)*

```python
person = object.__new__(Person)
print(person.__dict__)
person.__init__("John")
print(person.__dict__)
```

```
{}
{'name': 'John'}
```

```python
class Animal:
    def __new__(cls, type):
        print(f"Creating a new {cls.__name__} object...")
        return super().__new__(cls)

    def __init__(self, type):
        print(f"Initializing the animal object...")
        self.type = type


animal = Animal("Perro")
print(animal.__dict__)
```

```
Creating a new Animal object...
Initializing the animal object...
{'name': 'Perro'}
```

Clases Conceptos básicos - métodos *__repr__()* vs *__str__()*

- Aparentemente son iguales, pero podemos ver las diferencias si nos atenemos a su definición

- __repr__() : Crea la representación **oficial** de un objeto.
  Invocado por el método repr( )
  Devuelve cualquier expresión válida de Python p ej un diccionario

```
50  x="Hello World"
51  print(x.__str__())
52  print(x.__repr__())
```

```
Hello World
'Hello World'
```

- __str__() : Crea una representación **informal** de un objeto
  Invocado por str( ), print lo invoca implícitamente
  Si la clase no lo tiene implementado, print usa __repr__
  Siempre devuelve un string

```
54  class PrintThis:
55      def __init__(self):
56          self.greet = "Hola"
57          self.audience = "Coruña"
58
59  obj = PrintThis()
60  print(obj)
```

```
<__main__.PrintThis object at 0x7f398a15a710>
```

```
54  class PrintThis:
55      def __init__(self):
56          self.greet = "Hola"
57          self.audience = "Coruña"
58
59      def __str__(self):
60          return f"{self.greet} {self.audience}"
61
62  obj = PrintThis()
63  print(obj)
```

```
Hola Coruña
```

Clases Conceptos básicos - método *__eq__()*

- `__eq__()`: **Operador de igualdad binaria.**
  Se ejecuta cuando utilizamos el operador ==

```python
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.age == other.age

        elif isinstance(other, int):
            return self.age == other

        return False

john = Person('John', 'Doe', 25)
jane = Person('Jane', 'Doe', 25)
mary = Person('Mary', 'Doe', 27)

print(john == jane)
print(john == mary)
print(john == 25)
```

```
True
False
True
```

Dataclass vs regular custom class

```python
class RegularNBAPlayer:
    def __init__(self, name, last_name, number):
        self.name = name
        self.last_name = last_name
        self.number = number

    def __repr__(self):
        return (
            f"{self.__class__.__name__}"
            f"(name={self.name}, last name={self.last_name})"
        )

    def __eq__(self, other):
        if other.__class__ is not self.__class__:
            return NotImplemented
        return (self.name, self.last_name, self.number) == (
            other.name,
            other.last_name,
            other.number,
        )

king = RegularNBAPlayer("Lebron", "James", 23)
durantula = RegularNBAPlayer("Kevin", "Durant", 35)

print(king)
print(durantula)
print(king == durantula)
```

```
RegularNBAPlayer(name=Lebron, last name=James)
RegularNBAPlayer(name=Kevin, last name=Durant)
False
```

```python
from dataclasses import dataclass

@dataclass
class NBAPlayer:
    name: str
    last_name: str
    number: int

king =NBAPlayer("Lebron", "James", 23)
durantula = NBAPlayer("Kevin", "Durant", 35)

print(king)
print(durantula)
print(king == durantula)
```

```
NBAPlayer(name='Lebron', last_name='James', number=23)
NBAPlayer(name='Kevin', last_name='Durant', number=35)
False
```

Dataclass: Default values y Herencia

Podemos pasar valores por defecto, que tomarán si no les pasamos nada en el momento de crear la instancia

Pueden sobreescribirse y la única restricción es el orden: los opcionales no pueden ir después
de los obligatorios:
TypeError: non-default argument '' follows default argument

Hay un fix para esto > 3.10
@dataclass(kw_only=True)
Pero hace que obligatoriamente cuando instancias
Tengas que meter los argumentos como kw

```python
from dataclasses import dataclass

@dataclass
class Currency:
    value: float = 0.0
    exchange_rate: float = 0.17

@dataclass
class AUD(Currency):
    code: str = "AUD"
    name: str = "Australian Dollar"
    symbol: str = "A$"
    description: str = "AUD (A$) - Australian Dollar"

@dataclass(kw_only=True)
class USD(Currency):
    code: str = "USD"
    is_legal: bool
    name: str = "American Dollar"
    symbol: str = "USD$"
    description: str = "USD (USD$) - American Dollar"

default_currency = AUD()
non_default_currency = AUD(value=1.0, exchange_rate="0.19")
dollar = USD(is_legal=True)
```

```
AUD(value=0.0, exchange_rate=0.17, is_legal=True, code='AUD', name='Australian Dollar',...)
AUD(value=1.0, exchange_rate='0.19', is_legal=True, code='AUD', name='Australian Dollar',...)
USD(value=0.0, exchange_rate=0.17, code='USD', is_legal=True, name='American Dollar', ...) Am
```

Dataclass: frozen and order

Order indica a la Dataclass que se establece un patrón de jerarquía entre objetos, y que puede implementar los métodos <(=),(=)>

Frozen hace la dataclass inmutable, de forma que si intentamos asignar un valor post creación de la instancia, dará error

```python
from dataclasses import dataclass

@dataclass(order=True)
class Person():
    name: str
    age: int
    email: str

joe = Person('Joe', 25, 'joe@gmail.com')
mary = Person('Mary', 43, 'mary@gmail.io')
print(joe > mary)
```

```
False
```

```python
@dataclass(frozen=True)
class FrozenPerson():
    age: int
    email: str
    name: str = "Joe"

frozen_joe = FrozenPerson( 25, 'joe@gmail.com')
frozen_joe.name = "Juan"
```

```
dataclasses.FrozenInstanceError: cannot assign to field 'name'
```

Dataclass: field y post_init - Ejemplo de ordenación

field nos permite personalizar uno a uno los atributos de la dataclass.

Algunos valores:

- init, repr, compare: bool
- default_factory: callable
- default

__post_init__() nos permite implementar un __init__ clásico, si por ejemplo algún atributo necesita realizar un cálculo antes de asignarse

```python
from dataclasses import dataclass, field

@dataclass(order=True)
class Person():
    sort_index: int = field(init=False, repr=False)
    name: str
    age: int
    height: float
    email: str

    def __post_init__(self):
        self.sort_index = self.age

joe = Person('Joe', 45, 1.85, 'joe@gmail.com')
mary = Person('Mary', 43, 1.67, 'mary@gmail.com')

print(joe > mary)
```

Dataclass - Refactorizando code smells

Resumiendo lo visto hasta ahora:

- Son implementaciones de clases que nos “regalan” funcionalidades
- Podemos asignar valores por defecto
- Tenemos formas más complejas de inicialización

Podemos intentar refactorizar esas estructuras de código en formas de dict o tuple que todos nos hemos encontrado alguna vez y no sabemos de dónde salen

Los beneficios son obvios:

- Mayor legibilidad
- Podemos usar herencia!
- Agrupamos funciones en métodos de clase o instancia
- Podemos decorar estas clases

Referencias

Explicación paso a paso:

- https://realpython.com/python-data-classes/

Charla Pycon 2018:

- https://www.youtube.com/watch?v=T-TwcmT6Rcw , *Raymond Hettinger - Dataclasses: The code generator to end all code generators - PyCon 2018*