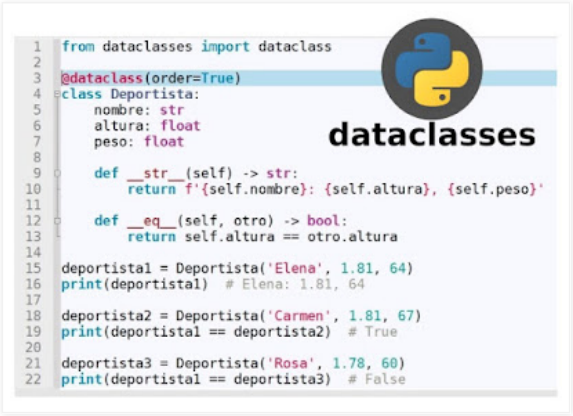
Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

martes, 12 de mayo de 2020

Data Classes: Clases de datos



Una de las características más interesantes de Python 3.7 es el soporte que proporciona el módulo **dataclasses** con el decorador **dataclass** para escribir clases de datos.

En una clase de datos se generan automáticamente algunos métodos especiales para clases simples. Los nombres de estos métodos, también llamados métodos mágicos, comienzan y finalizan con un doble subrayado como **__init__()**, **__repr__()**, **__eq__()**, entre otros.

Como es sabido el método **__init__()** se utiliza en una clase para inicializar un objeto y se invoca sin hacer una llamada específica, simplemente, cuando se instancia una clase. De ahí, que se le conozca como método constructor.

De modo que escribir una clase como la del siguiente ejemplo era lo normal hasta hace muy poco. En este caso la acción de instanciar la clase para crear un objeto lleva implícita la llamada al método **__init__()** que efectúa las asignaciones de **nombre**, **altura** y **peso**. Por ello, cuando se imprime la **altura** se obtiene el valor asignado sin que sea necesario hacer nada más:

```
class Deportista:
    def __init__(self, nombre, altura, peso):
        self.nombre = nombre
        self.altura = altura
        self.peso = peso

deportista1 = Deportista('Elena', 1.81, 64)
print(deportista1.altura)  # 1.81
```

Bien, la nueva característica que comentamos permite ahora escribir la clase anterior de forma más simplificada y clara:

```
from dataclasses import dataclass

@dataclass
class Deportista:
    nombre: str
    altura: float
    peso: float

deportista1 = Deportista('Elena', 1.81, 64)
print(deportista1.altura)  # 1.81
```

Como puede observarse a la clase **Deportista** le precede el decorador **dataclass** y no tiene definido el método **__init__()**.

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

Una de las funciones del decorador es localizar las variables de clase que llevan anotaciones de tipos para conocer los campos que tiene la clase de datos. Después, con respecto al modo de instanciar la clase no se advierte ningún cambio con respecto al uso habitual.

Los métodos de dataclass

La magia obviamente está en el decorador de clase que ayuda a reducir el código porque no solo genera el método **__init__()**, también hace lo propio con los métodos **__str__()**, **__repr__()** y, opcionalmente, con algunos métodos más.

Y sabemos que el decorador genera el método **__str__()** (que devuelve una cadena con una representación legible de los datos) porque es llamado cuando se imprime el objeto o cuando se hace uso de la función **str()**:

```
print(deportita1)  # Deportista(nombre='Elena', altura=1.81, peso=64)

atleta = str(deportista1)
print(atleta)  # Deportista(nombre='Elena', altura=1.81, peso=64)
```

Algunos de estos métodos también pueden reescribirse dentro de la clase para modificar su comportamiento predeterminado. En el ejemplo siguiente el método **__str__()** se ha reescrito y devuelve una cadena con el siguiente formato: **'nombre: altura, peso'**

```
@dataclass
class Deportista:
    nombre: str
    altura: float
    peso: float

    def __str__(self) -> str:
        return f'{self.nombre}: {self.altura}, {self.peso}'

deportista1 = Deportista('Elena', 1.81, 64)
print(str(deportista1))  # Elena: 1.81, 64
```

Los parámetros de dataclass

El decorador **dataclass** cuenta también con varios parámetros para ajustar su funcionamiento:

```
@dataclass(init=True, repr=True, eq=True, order=False,
           unsafe_hash=False, frozen=False)
```

- **init**, **repr** y **eq**: Por defecto estos parámetros tienen el valor **True** para que el decorador genere los métodos **__init__()**, **__repr__()** y **__eq__()**, respectivamente, aunque si la clase los redefine serán ignorados.
- **order**: Por defecto tiene el valor **False** pero si se establece a **True** el decorador generará los métodos especiales **__gt__()**, **__ge__()**, **__lt__()** y **__le__()**. En este caso no se permite la reescritura, por lo que si la clase redefine alguno de ellos se producirá una excepción.
- **unsafe_hash**: Por defecto tiene el valor **False** y en este caso el decorador generará el método **__hash__()** de acuerdo a la configuración que tengan los parámetros **eq** y **frozen**.
- **frozen**: Por defecto tiene el valor **False** pero si se establece a **True** cualquier intento de asignación a los campos producirá una excepción.

En el siguiente ejemplo se establece el parámetro **order** con el valor **True** para que el decorador **dataclass** genere los métodos **__gt__()**, **__ge__()**, **__lt__()** y **__le__()** que se corresponden con las comparaciones "*mayor que*", "*mayor o igual que*", "*menor que*" y "*menor o igual que*", respectivamente.

Las variables de clase son inicializadas cuando los objetos se crean omitiendo dichos valores. En este ejemplo se crean tres objetos asignando un valor al campo **peso** para realizar comparaciones y conocer si el valor del campo en un objeto es "*mayor que*" en otro. Y sabemos que el método **__gt__()** se ha generado porque es llamado cuando se comparan los objetos con el operador "**>**":

```
@dataclass(order=True)
class Deportista:
    nombre: str = 'Desconocido'
    altura: float = 0
    peso: float = 0

deportista1 = Deportista(peso=64)
deportista2 = Deportista(peso=62)
deportista3 = Deportista(peso=67)
```

simultáneamente varias operaciones en el mismo espacio de proceso se...

Archivo

mayo 2020 (1)

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

```
print(deportista1 > deportista2)  # True
print(deportista1 > deportista3)  # False
```

Ahora es suficiente con cambiar el valor de **order** a **False** para verificar que en ese caso los métodos no están disponibles y que se produce una excepción porque la comparación "*mayor que*" no estaría soportada por la clase.

En el ejemplo siguiente se establece el parámetro **frozen** a **True** con lo cual es posible instanciar la clase para crear objetos pero no es posible asignar valores porque el objeto ha sido "congelado". El intento de asignación produce una excepción de tipo **dataclasses.FrozenInstanceError**:

```
@dataclass(frozen=True)
class Deportista:
    nombre: str = 'Desconocido'
    altura: float = 0
    peso: float = 0

deportista1 = Deportista(peso=64)
deportista1.peso = 63  # dataclasses.FrozenInstanceError
```

La función asdict()

La función **asdict()** se utiliza para convertir una instancia de clase de datos en un diccionario Python.

En el ejemplo siguiente se importa la función **asdict** que se emplea para convertir el objeto **deportista1** en un diccionario usando los campos de la clase de datos para definir sus claves y sus valores:

```
from dataclasses import dataclass, asdict

@dataclass
class Deportista:
    nombre: str
    altura: float
    peso: float

deportista1 = Deportista('Elena', 1.81, 64)
dicc1 = asdict(deportista1)

if dicc1['altura'] > 1.75:
   print(dicc1['nombre'], 'supera la altura')
```

La función field()

La función **field()** permite facilitar información adicional al decorador relativa a cada campo que la utilizará en la generación de los métodos.

En el ejemplo que sigue para el atributo **peso** se establecen los parámetros **init** y **repr** a **False**. Esto indica al decorador que el objeto podrá crearse sin el atributo **peso** y que cuando se imprima su representación será omitida esta información. No obstante, como el atributo **peso** existe se le podrá asignar un valor en cualquier momento y acceder al mismo después de la asignación.

```
from dataclasses import dataclass, field

@dataclass
class Deportista:
    nombre: str
    altura: float
    peso: float = field(init=False, repr=False)

deportista1 = Deportista('Elena', 1.81)
deportista1.peso = 64
print(deportista1)  # Deportista(nombre='Elena', altura=1.81)
print(deportista1.peso)  # 64
```

Herencia

Las clases de datos también pueden heredar atributos y métodos de otras clases de datos.

En el siguiente ejemplo la clase de datos **Equipo** hereda de la clase **Deportista** sus variables y métodos aunque en esta ocasión ambas clases redefinen el método **__str__()** para que al ser

llamado muestre información diferente en cada ámbito.

En la clase que hereda, **Equipo**, la variable **equipo** debe tener un valor por defecto para que cuando se instancie la clase **Deportista** no se produzca una excepción de tipo **TypeError**. Esto es así, aún cuando el atributo **equipo** queda fuera del alcance de la clase **Deportista**.

```
from dataclasses import dataclass

@dataclass
class Deportista:
    nombre: str
    altura: float = 0
    peso: float = 0

    def __str__(self) -> str:
        return f'{self.nombre}: {self.altura}, {self.peso}'

@dataclass
class Equipo(Deportista):
    equipo: str = 'desconocido'

    def __str__(self) -> str:
        return f'{self.nombre}: {self.equipo}'

# Instancia la clase Deportista para crear objeto:
deportista1 = Deportista('Elena', 1.81, 64)

# Imprime llamando al método __str__() de
# la clase Deportista:
print(deportista1)  # Elena: 1.81, 64

# Instancia la clase Equipo para crear objeto:
deportista2 = Equipo('Marta', equipo='Sevilla')

# Imprime llamando al método __str__() de
# la clase Equipo:
print(deportista2)  # Marta: Sevilla

# Asigna valores a atributos de objeto de la clase Equipo:
deportista2.altura = 1.76
deportista2.peso = 68

# Imprime representación formal de objeto de la clase Equipo:
print(repr(deportista2))

# Equipo(nombre='Marta', altura=1.76, peso=68, equipo='Sevilla')
```

Relacionado:

- Anotaciones de datos: typing
- Programación funcional: funciones de orden superior (la función decorador)

Publicado por Pherkad en 14:06

Etiquetas: Python3

Inicio Entrada antigua

2014-2020 | Alejandro Suárez Lamadrid y Antonio Suárez Jiménez, Andalucía - España
. Tema Sencillo. Con la tecnología de Blogger.