clase_Python_type_hints

September 1, 2022

1 Este material está sacado del Seminario de Python

2 Control de tipos en Python

3 Antes de empezar: ¿para qué nos sirve conocer el tipo de datos de una variable?

- Los tipos de datos nos permiten relacionar un **conjunto de valores** que son de ese tipo con las **operaciones** que se pueden aplicar sobre esos valores.

4 ¿Qué es un sistema de tipos?

- El sistema de tipos es un conjunto de reglas que tiene un lenguaje que nos permite manipular los datos de nuestros programas.
- Incluyen las conversiones explícitas e implícitas que podemos realizar.

5 Lenguajes con tipado estático vs. dinámico

- Se refiere a si el tipo de una variable se conoce en tiempo de compilación o en ejecución.

[ ]:
```
x = "casa"
type(x)
```

6 Lenguajes fuertemente tipados vs. débilmente tipados

- **Fuertemente tipados**: no se puede usar aplicar operaciones de otro tipo a menos que se haga una conversión explícita. Por ejemplo: Java, Pascal y Python.
- **Débilmente tipados**: se pueden mezclar en una misma expresión valores de distinto tipo. Por ejemplo PHP y Javascript.

```
x = "a" + 5
```

[ ]:
```
x = "a" + 5
x
```

7 Python

- Es un lenguaje **fuertemente tipado**.
- Posee un **tipado dinámico**: el intérprete de Python realiza el chequeo de tipos durante la ejecución y el tipo de una variable puede cambiar durante su tiempo de vida.

8 La verificación de tipos

- Se refiere a chequeo de tipos.
- Es donde se aplican las reglas definidas en el sistema de tipos.
- La verificación de tipos puede ser:
  - estática: ocurre en tiempo de **compilación**. Por ejemplo: Pascal y C
  - dinámica: ocurre en tiempo de **ejecución**. Por ejemplo PHP, Ruby y Python.

[ ]:
```
opcion = input("ingresa 1 para verificar y 2 para no")
if opcion == "1":
    print("Estoy chequeando...")
    print("e" + 4 )
else:
    print("Ahora no estoy dando error")
```

9 Duck Typing

9.1 "Si parece un pato, nada como un pato y suena como un pato, entonces probablemente sea un pato"

10 Observemos el siguiente código

- Sacado de https://realpython.com/python-type-checking/

[ ]:
```
def headline(text, align=True):
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "-")
```

[ ]:
```
print(headline("python type checking"))
```

[ ]:
```
print(headline("python type checking", align=False))
```

11 Probemos esto:

[ ]:
```
print(headline("python type checking", align="left"))
```

12 Python permite agregar sugerencias de tipos: anotaciones

[ ]:
```
def headline(text: str, align: bool = True) -> str:
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "-")
```

[ ]:
```
print(headline("python type checking", align="left"))
```

- ¿Cambió algo?

Si bien estas anotaciones están disponibles en tiempo de ejecución a través del atributo **___annotations___**, no se realiza ninguna verificación de tipo en tiempo de ejecución.

[ ]:
```
headline.__annotations__
```

13 Pero si lo abrimos en un IDE (PyCharm en este caso)

- Usamos un verificador de tipos externo.
- La herramienta más común para realizar la verificación de tipos es Mypy

14 mypy

- Se instala con pip: pip install mypy

15 ¿Cómo resolvemos este "error"?

[ ]:
```
def headline(text: str, centered: bool = True) -> str:
    if centered:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "-")
```

[ ]:
```
print(headline("python type checking"))
print(headline("use mypy", centered=True))
```

16 Anotaciones

- Como vimos, en las funciones se puede agregar anotaciones sobre los argumentos y el valor de retorno.
- En general:

```
def funcion(arg1: arg_type, arg2: arg_type = valor) -> return_type:
    ...
```

[ ]:
```python
import math

def area_circunferencia(radio: float) -> float:
    return math.pi * radio ** 2

area = area_circunferencia(2)
print(area)
```

17 También se pueden hacer anotaciones de variables

[ ]:
```python
pi: float = 3.1415

def area_circunferencia(radio: float) -> float:
    return math.pi * radio ** 2

area = area_circunferencia(2)
print(area)
```

[ ]:
```python
area_circunferencia.__annotations__
```

[ ]:
```python
__annotations__
```

18 Un poco más sobre anotaciones

- Se puede realizar una anotación de una variable **sin darle un valor**.

[ ]:
```python
mensaje: str
__annotations__
```

[ ]:
```python
mensaje = 10
mensaje
```

19 Otros ejemplos

[ ]:
```python
nombre_bandas: list = ["Led Zeppelin", "AC/DC", "Queen"]
notas: tuple = (7, 8, 9, 10)
opciones: dict = {"centered": False, "capitalize": True}
```

- ¿Cómo podemos indicar que se trata de una lista de elementos str? ¿O una tupla de enteros?

20 El modulo typing

- Permite escribir anotaciones un poco más complejas.

[ ]:
```python
from typing import Dict, List, Tuple

nombre_bandas: List[str] = ["Led Zeppelin", "AC/DC", "Queen"]
notas: Tuple[int, int, int, int] = (7, 8, 9, 10)
opciones: Dict[str, bool] = {"centered": False, "capitalize": True}
```

21 Veamos este otro ejemplo

[ ]:
```python
from typing import List, Sequence

def cuadrados(elems: Sequence[float]) -> List[float]:
    return [x**2 for x in elems]
```

[ ]:
```python
cuadrados([1, 2, 3])
```

- Una secuencia es cualquier objeto que admita **len ()** y **___ getitem ___ ()**, independientemente de su tipo real.

22 ¿Qué pasa con este código?

[ ]:
```python
import random

def elijo_al_azar(lista_de_elementos):
    return random.choice(lista_de_elementos)

lista = [1, "dos", 3.1415]
elijo_al_azar(lista)
```

22.0.1 Para incorporar las anotaciones usamos el tipo: Any

[ ]:
```python
import random
from typing import Any, Sequence

def elijo_al_azar(lista_de_elementos: Sequence[Any]) -> Any:
    return random.choice(lista_de_elementos)

lista = [1, "dos", 3.1415]
elijo_al_azar(lista)
```

23 Anotaciones y POO

23.1 ¿Cómo agregamos anotaciones a los métodos?

[ ]:
```
class Jugador:

    def __init__(self,
                 nombre: str,
                 juego: str = "Tetris",
                 tiene_equipo: bool = False,
                 equipo: str = None) -> None:

            self.nombre = nombre
            self.juego = juego
            self.tiene_equipo = tiene_equipo
            self.equipo = equipo

    def jugar(self) -> None:
            if self.tiene_equipo:
                    print (f"{self.nombre} juega en el equipo {self.equipo} al␣
 ↪{self.juego}")
            else:
                    print(f"{self.nombre} juega solo al {self.juego}")
```

- Se usan las mismas reglas que para las funciones.
- **self** no necesita ser anotado. ¿De qué tipo es?

23.2 ¿Cómo agregamos anotaciones a las variables de instancia y de clase?

- Se usan las mismas reglas que para las variables comunes.

[ ]:
```
class SuperHeroe():
    """ Esta clase  define a un superheroe
    villanos:  representa a los enemigos de todos los superhéroes
    """
    villanos: List[str] = []

    def __init__(self, nombre: str, alias: str) -> None:
                self._nombre = nombre
                self._enemigos = []
```

24 Hasta acá llegamos...

24.1 Más info

- La PEP 3107 introdujo la sintaxis para las anotaciones de funciones, pero la semántica se dejó deliberadamente sin definir.

- La PEP 484 introduce un módulo provisional para proporcionar definiciones y herramientas estándares, junto con algunas convenciones para situaciones en las que las anotaciones no están disponibles.
- La PEP 526: tiene como objetivo mostrar de qué manera se pueden realizar anotación de varoables (incluidas las variables de clase y las variables de instancia),
- Artículo de RealPython: https://realpython.com/python-type-checking/
- Artículo de the state of type hints in Python de Bernát Gábor.