Tabla de contenidos

1. Motivación
2. Anotaciones
3. Módulo `typing`
4. Listas y `collections`
5. Clases y autorreferencia

Motivación

A medida que nuestros programas crecen en complejidad y tamaño, resulta cada vez más difícil recordar que tipos de datos usan las distintas variables y funciones a lo largo de los módulos que hemos creado. Esto es peor cuando debemos hacer mantención a un código que no hemos trabajado por meses, o de otra persona. Una alternativa muy recomendada es el uso de comentarios para documentar las particularidades de un código. Esto ayuda enormemente la comprensión posterior de un código.

Cuando programamos, para mantener el orden en el código, muchas veces decidimos utilizar un único tipo de dato para una variable determinada. Por ejemplo, una variable llamada `nombre` puede ser de cualquier tipo, pero la utilizamos solo como un `string`. Lo mismo ocurre con argumentos, atributos y retornos de funciones. Esta información es muy útil al momento de documentar, pero también ayuda a que las *IDE*s a que sugieran código y alerten posibles errores. Por esto Python nos ofrece una manera especial de incorporarla al código, las anotaciones.

Anotaciones

Python nos permite declarar el tipo esperado de una variable, argumento o atributo por medio de anotaciones. Para esto, después del nombre, debemos escribir `:` y el tipo. Por ejemplo, podemos declarar que la intención de una variable es almacenar solamente números enteros de la siguiente forma:

```
variable_numerica: int = 42
```

Lo mismo se puede aplicar para una variable que es un string:

```
nombre: str = "Julián"
```

Para los casos de funciones o métodos, podemos anotar el tipo de datos de los argumentos. Para esto, se hace del mismo modo que una variable.

```
def direccion(calle: str, numero: int):
    return f'{calle.strip().title()} {numero}'
```

Además, podemos anotar el tipo de dato esperado del retorno. Para esto, después de la declaración de los argumentos debemos escribir `->` y el tipo:

```
def direccion(calle: str, numero: int) -> str:
    return f'{calle.strip().title()} {numero}'
```

De este modo, podemos mantener un cierto grado de documentación funcional en nuestro código. Ya que las *IDE*s pueden analizar las anotaciones, dar sugerencias de atributos y métodos mientras se programa, y levantar alertas si se escribe algo que no cumpla con los tipos ya declarados:

```
variable_numerica = "la respuesta"  # Alerta IDE: Expected type 'int', got 'srt' instead

direccion("Alameda", 10.0)  # Alerta IDE: Expected type 'int', got 'float' instead
```

```
'Alameda 10.0'
```

Finalmente, podemos utilizar nuestras propias clases como tipo de dato para las anotaciones. En el siguiente ejemplo se anotó que `variable` será una instancia del tipo `Computador`:

```
class Computador:
    def __init__(self) -> None:
        self.energia: int = 100

    def reducir_energia(self) -> None:
        self.energia -= 1


variable: Computador = Computador()
```

Cabe destacar que las anotaciones y alertas de las *IDE*s no impiden que un código se ejecute.

Módulo `typing`

Para extender y flexibilizar las anotaciones, el módulo `typing` de Python nos provee de tipos adicionales. Algunos de los más útiles son:

- `Any`: Nos permite declarar que el tipo de una variable no está restringida:

```
from typing import Any

variable: Any = None
variable = 1
variable = 2.0
variable = "3"
```

- `Union`: Nos permite declarar que una variable puede ser de más de un tipo distinto. Por ejemplo, si va a ser un número, independiente de si es entero o real, podemos usar: `Union[int, float]`. Desde python 3.10 también podemos usar el operador `|`, siendo el ejemplo anterior equivalente a `int | float`:

```
from typing import Union


def dividir_numeros(numerador: Union[int, float], denominador: Union[int, float]) -> float:
    return numerador / denominador
```

- `Callable`: Nos permite declarar que una variable contiene la referencia a una función. Si lo deseamos, también podemos especificar los tipos de los argumentos y del retorno: `Callable[[TipoArgumento, ...], TipoRetorno]`:

```
from typing import Callable
```

```
funcion_dividir: Callable[[Union[int, float], Union[int, float]], float] = dividir_numeros
funcion_dividir(1.0, 2)
```

```
0.5
```

- `Optional`: Nos permite declarar que una variable es de algún tipo o `None`. Escribir `Optional[X]` es equivalente a `Union[X, None]` y `X | None`:

```
from typing import Optional


variable: Optional[int] = None
variable = 1
```

Listas y `collections`

Python ofrece estructuras de datos cuyo objetivo es hacer de contenedores para otros elementos, algunos *built-in* y otros por medio del módulo `collections`. De estas estructuras, actualmente conocen las listas que son del tipo `list`.

Para poder definir el tipo de elementos contenidos en una lista, el módulo `typing` nos ofrece `List`. Por ejemplo, podemos declarar que una variable corresponde a una lista que solo contiene números enteros usando `List[int]`:

```
from typing import List


lista: List[int] = [1, 2, 3, 4, 5, 6]
```

De este mismo modo, el módulo `typing` ofrece tipos para el resto de las estructuras de datos *built-in* y del módulo `collections`. Les recomendamos revisar la documentación de Python si desean utilizarlas.

Clases y autorreferencia

Un problema es que, con lo visto anteriormente, no podremos usar un tipo personalizado en la definición de su propia clase. Esto es porque al momento de definir la clase su tipo aún no existe.

```
from typing import Optional


class Persona:
    def __init__(self: Persona) -> None:
        self.bff: Optional[Persona] = None

    def asignar_bff(self: Persona, otro: Persona) -> None:
        self.bff = otro
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[12], line 4
      1 from typing import Optional
----> 4 class Persona:
```

```
      5     def __init__(self: Persona) -> None:
      6         self.bff: Optional[Persona] = None

Cell In[12], line 5, in Persona()
      4 class Persona:
----> 5     def __init__(self: Persona) -> None:
      6         self.bff: Optional[Persona] = None
      8     def asignar_bff(self: Persona, otro: Persona) -> None:

NameError: name 'Persona' is not defined
```

Una solución, hasta ahora, es postergar la evaluación de las anotaciones, de tal modo que al momento de ser evaluadas todos los tipos ya existan. Para hacerlo solo es necesario importar annotations del módulo `__future__`.

```python
from __future__ import annotations  # Esto habilita la evaluación postergada de las anotaciones
from typing import Optional


class NuevaPersona:
    def __init__(self: NuevaPersona) -> None:
        self.bff: Optional[NuevaPersona] = None

    def asignar_bff(self: NuevaPersona, otro: NuevaPersona) -> None:
        self.bff = otro
```