

Anotaciones de tipos o signaturas

En algunos lenguajes de programación, los tipos de datos de sus datos y funciones se declaran explícitamente, Por ejemplo, en C++:

```
float area(int radio, float PI=3.1416) {  
    return PI*pow(radio, 2);  
}  
  
int main() {  
    float radio = 4.5;  
    float resultado = area(radio);  
    cout << "El área de un círculo de radio " << radio << " es " <<  
    resultado << endl;  
}
```

Esto permite realizar las comprobaciones de tipos *en tiempo de compilación* de un programa, por lo que se suele decir que son lenguajes *tipados estáticamente*.

En Python, los tipos de las variables, funciones y argumentos no se declaran. El tipo de datos de cada variable o función se va calculando *dinámicamente*, es decir, durante su ejecución.

Sin mencionar ahora las ventajas de esta decisión, presenta dos inconvenientes:

- El diseño de programas erróneos desde el punto de vista de los tipos es posible, y pasa fácilmente desapercibido hasta el momento de su ejecución porque no hay una comprobación antes de dicha ejecución.
- La ausencia de información sobre los tipos ayuda a camuflar dichos posibles errores en la coherencia de los tipos de los datos.

Para paliar este segundo inconveniente, en Python se recomienda con mayor fuerza incluir una documentación de forma estándar, el *docstring*, en la cual existen secciones especiales para especificar la signatura de una función, es decir, los tipos de los parámetros y del resultado de la misma. Este asunto se trata en otros documentos. Damos aquí una referencia disponible en Internet:

<https://python-para-impacientes.blogspot.com/2014/02/docstrings.html> (<https://python-para-impacientes.blogspot.com/2014/02/docstrings.html>)

También, es posible añadir *anotaciones* de tipos. Y desde Python 3.5, la librería estándar incluye el módulo `typing`. Aunque las anotaciones no son operativas, esta característica permite el desarrollo de herramientas que realizan este chequeo, y de hecho existen actualmente *herramientas de control de tipos* que pueden usarse antes de la ejecución de un programa. De este modo, se reduce el primer inconveniente mencionado. Entre estas herramientas, mencionamos `mypy`, `pyre-check` y `pytype`.

Anotaciones de variables, constantes y funciones

He aquí las primeras anotaciones de tipos aplicadas a variables, constantes y funciones:

```
In [1]: ▶ n: int = 37
frase: str = "El mundo era tan reciente que muchas cosas carecían de nom
print(n, frase)

def area_circulo(radio: float) -> float:
    Pi: float = 3.14
    return Pi*radio**2

radio: float = 4.5
resultado: float = area_circulo(radio)
print("El área de un círculo de radio ", radio, " es " , resultado)

37 El mundo era tan reciente que muchas cosas carecían de nombre
El área de un círculo de radio 4.5 es 63.585
```

Las anotaciones de tipos asociadas a una función pueden recuperarse mediante el método `__annotations__`, que genera un diccionario. Si aún no has visto este tipo de estructura, no te preocupes: observa a continuación su uso, que es muy intuitivo.

Para recuperar el tipo declarado de una función, se ha de especificar el nombre de la función:

```
In [2]: ▶ # Podemos inspeccionar las variables y constantes

print(__annotations__)
print(area_circulo.__annotations__)

print(__annotations__["n"])
print(type(n))

{'n': <class 'int'>, 'frase': <class 'str'>, 'radio': <class 'float'>,
'resultado': <class 'float'>}
{'radio': <class 'float'>, 'return': <class 'float'>}
<class 'int'>
<class 'int'>
```

Las anotaciones de tipos no se comprueban

Las anotaciones de tipos únicamente especifican el tipo de las variables y el tipo que se espera de los parámetros y resultados de las funciones. Pero únicamente tienen un valor indicativo, pues Python no las tiene en cuenta en realidad, y **no hace las comprobaciones de tipos**.

```
In [3]: ▶ def diez_veces(n: int) -> int:
        return n*10

        print(diez_veces(5))
        print(diez_veces(5.0))
        print(diez_veces("5"))
        print(diez_veces([1, 2, 3]))

        n: int = "345"
        print(diez_veces(n))
```

```
50
50.0
5555555555
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2,
3, 1, 2, 3, 1, 2, 3]
345345345345345345345345345345345345345345345345345345
```

Podríamos hacer incluso anotaciones incoherentes y Python no las detectaría hasta que se produjera un error, esto es, en tiempo de ejecución:

```
In [4]: ▶ def suma(a: int, b: str) -> int:
        return a+b

        print(suma(4, 5))
        print(suma(4, "caracol"))
```

```
9
```

```
-----
----
TypeError                                Traceback (most recent call 1
ast)
Input In [4], in <cell line: 5>()
      2     return a+b
      4 print(suma(4, 5))
----> 5 print(suma(4, "caracol"))

Input In [4], in suma(a, b)
      1 def suma(a: int, b: str) -> int:
----> 2     return a+b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Anotaciones con identificadores estándar

Podemos usar como anotaciones los nombres de los tipos predefinidos: `int` , `float` , `complex` , `str` , `bool` , `tuple` , `list` , `dict` :

```
In [5]: ▶ n: int = 7
e: float = 2.71828
z: complex = 1-5.0j
afirmativo: bool = True
frase: str = "El lagarto está llorando, la lagarta está llorando..."
par: tuple = (2, 3)
lista: list = [1, 2, 3]
dicc: dict = {"C": 34, "A": 23}

print(n, e, z, afirmativo)
print(frase)
print(par, lista)
print(dicc)

print(".....")

tipos = __annotations__
print("n: -> ", tipos["n"])
print("e: -> ", tipos["n"])
print("z: -> ", tipos["n"])
print("afirmativo: -> ", tipos["afirmativo"])
print("frase: -> ", tipos["frase"])
print("par: -> ", tipos["par"])
print("lista: -> ", tipos["lista"])
print("dicc: -> ", tipos["dicc"])

7 2.71828 (1-5j) True
El lagarto está llorando, la lagarta está llorando...
(2, 3) [1, 2, 3]
{'C': 34, 'A': 23}
.....
n: -> <class 'int'>
e: -> <class 'int'>
z: -> <class 'int'>
afirmativo: -> <class 'bool'>
frase: -> <class 'str'>
par: -> <class 'tuple'>
lista: -> <class 'list'>
dicc: -> <class 'dict'>
```

También podemos introducir ad hoc nombres nuevos de tipos de datos inexistentes, a título informativo, o durante el proceso de desarrollo, de manera provisional, mientras definimos con mayor precisión un tipo de datos.

```
In [6]: ▶ def mi_fun(n: "tipo_a", m: list) -> "tipo_c":
        return (n, m)

print(mi_fun.__annotations__)

{'n': 'tipo_a', 'm': <class 'list'>, 'return': 'tipo_c'}
```

Anotaciones con clases definidas por el usuario

También podemos usar anotaciones en los métodos de las clases definidas por el usuario, y también los identificadores de las clases:

```
In [7]: ▶ class VectorR2:
    def __init__(self, x: float, y: float):
        self._x = x
        self._y = y
    def __str__(self):
        return "<" + str(self._x) + ", " + str(self._y) + ">"

def alargar(v: VectorR2, k: float) -> VectorR2:
    return VectorR2(v._x * k, v._y * k)

u = VectorR2(3, 4)
v = alargar(u, 1.5)

print(u, v)

print(alargar.__annotations__)

<3, 4> <4.5, 6.0>
{'v': <class '__main__.VectorR2'>, 'k': <class 'float'>, 'return': <class '__main__.VectorR2'>}
```

Anotaciones con estructuras de datos definidas por el usuario

En las estructuras de datos, se pueden dar también, opcionalmente, los tipos de sus componentes. Hay costumbre de usar distintos tipos de notación, aunque la verdad es que esta notación no es estándar, y existen otras propuestas alternativas para detallar los tipos de las estructuras de datos con los de sus componentes.

```
In [8]: ▶ # OJO: esta notación no es estándar

b: [int] = [1, 2, 3]
par: (int, str) = (2, "dos")
conj: {int} = {1, 2, 3}
dicc: {str: int} = {"C": 34, "A": 23}

print(b, par, conj, dicc)
print(__annotations__["b"])
print(__annotations__["par"])
print(__annotations__["conj"])
print(__annotations__["dicc"])

[1, 2, 3] (2, 'dos') {1, 2, 3} {'C': 34, 'A': 23}
[<class 'int'>]
(<class 'int'>, <class 'str'>)
{<class 'int'>}
{<class 'str'>: <class 'int'>}
```

In [9]:  *# OJO: La siguiente notación es más adecuada, pero tampoco es estándar:*

```
b: list[int] = [1, 2, 3]
par: tuple[int, str] = (2, "dos")
conj: set[int] = {1, 2, 3}
dicc: dict[str, int] = {"C": 34, "A": 23}

print(b, par, conj, dicc)
print(__annotations__["b"])
print(__annotations__["par"])
print(__annotations__["conj"])
print(__annotations__["dicc"])
```

```
[1, 2, 3] (2, 'dos') {1, 2, 3} {'C': 34, 'A': 23}
list[int]
tuple[int, str]
{<class 'int'>}
dict[str, int]
```

Anotaciones con typing

las notaciones anteriores son adecuadas cuando no necesitamos que Python haga comprobaciones de tipos. Pero la siguiente es más completa, aunque tampoco realizan comprobaciones de tipos. Se llama `typing`, y antes usarla se debe instalar la librería:

```
```pip install typing```
```

Hecho esto, podemos usarla de manera muy sencilla:

In [10]:  `from typing import List, Tuple, Dict, Set`

```
b: List[int] = [1, 2, 3]
par: Tuple[int, str] = (2, "dos")
conj: Set[int] = {1, 2, 3}
dicc: Dict[str, int] = {"C": 34, "A": 23}

print(b, par, conj, dicc)
print(__annotations__["b"])
print(__annotations__["par"])
print(__annotations__["conj"])
print(__annotations__["dicc"])
```

```
[1, 2, 3] (2, 'dos') {1, 2, 3} {'C': 34, 'A': 23}
typing.List[int]
typing.Tuple[int, str]
typing.Set[int]
typing.Dict[str, int]
```

He aquí un ejemplo de estructuras algo más complejas, anidando construcciones:

```
In [11]: ▶ # Tipos de estructuras complejas:

perros: List[Tuple[str, float, Set[str]]] = \
 [("Pipo", 0.75, {"caminar", "perseguir pájaros", "caramelos"}),
 ("Blacky", 0.35, {"dormir", "morder otros perros", "sus chuches"})]

print(__annotations__["perros"])

typing.List[typing.Tuple[str, float, typing.Set[str]]]
```

Los tipos de datos pueden nombrarse mediante *alias* de tipos, que no son más que identificadores de tipos definidos por el usuario, ya sea por tratarse de estructuras de datos más complejas o para facilitar la lectura de un programa.

```
In [12]: ▶ TipoPerro = Tuple[str, float, Set[str]]
ListaDePerros = List[TipoPerro]

perros: ListaDePerros = \
 [("Pipo", 0.75, {"caminar", "perseguir pájaros", "caramelos"}),
 ("Blacky", 0.35, {"dormir", "morder a otros perros", "sus chuches"})]

def estatura_perruna(p: TipoPerro) -> float:
 return p[1]

print(__annotations__["perros"])
print(estatura_perruna.__annotations__)

typing.List[typing.Tuple[str, float, typing.Set[str]]]
{'p': typing.Tuple[str, float, typing.Set[str]], 'return': <class 'float'>}
```

Pero sigue sin comprobar tipos, como ya decíamos:

```
In [13]: ▶ p: ListaDePerros = [1, 2, 3]
print(__annotations__["perros"])

typing.List[typing.Tuple[str, float, typing.Set[str]]]
```

Para la construcción de tipos, podemos usar una operación `Optional` y otra `Union` :

```
In [14]: ▶ # Tipo Opcional:

from typing import Optional

MaybeFloat = Optional[float]

def divide(a: int, b: int) -> MaybeFloat:
 if b != 0:
 return a / b

print(divide.__annotations__)
print(divide(6, 2))
print(divide(6, 0))

{'a': <class 'int'>, 'b': <class 'int'>, 'return': typing.Optional[float]}
```

3  
None

```
In [15]: ▶ # Tipo Unión:

from typing import Union

SolucionesEcuacion = Union[float, str, None]

def solve_ec_1_grado(a: float, b: float) -> SolucionesEcuacion:
 # solución de una ecuación de la forma ax + b = 0
 if a == 0:
 if b == 0:
 return "Infinitas soluciones"
 else:
 return None # No tiene solución
 else: # a != 0
 return -b/a

print(solve_ec_1_grado(2, 2))
print(solve_ec_1_grado(0, 2))
print(solve_ec_1_grado(0, 0))

-1.0
None
Infinitas soluciones
```

Para funciones como parámetros o resultados, tenemos el constructor de tipos `Callable` :

```
In [16]: ▶ from typing import Callable

Fun_R_R = Callable[[float], float]

def composition(f: Fun_R_R, g: Fun_R_R) -> Fun_R_R:
 return lambda x: f(g(x))

fg = composition(lambda x: x**2, lambda x: x+1)
print(fg(7))
```



## Comprobación de tipos de datos con mypy

Una herramienta muy útil de comprobación de tipos es `mypy`. Antes usarla se debe instalar la librería:

```
```pip install mypy```
```

Hecho esto, podemos usarla de manera muy sencilla:

```
In [17]: ▶ %%writefile diez_bien.py

def diez_veces(n: int) -> int:
    return n*10

n: int = 345
print(n)
```

Overwriting diez_bien.py

```
In [18]: ▶ ! mypy diez_bien.py
```

Success: no issues found in 1 source file

```
In [19]: ▶ %%writefile diez_mal.py

def diez_veces(n: int) -> int:
    return n*10

n: int = "345"
print(n)
```

Overwriting diez_mal.py

```
In [20]: ▶ ! mypy diez_mal.py
```

diez_mal.py:5: error: Incompatible types in assignment (expression has type "str", variable has type "int") [assignment]
Found 1 error in 1 file (checked 1 source file)

```
In [21]: ▶ %%writefile estructuras_arbitrarias_1.py
```

```
def suma_lista(lista: [int]) -> int:
    return sum(lista)
```

Overwriting estructuras_arbitrarias_1.py


```
In [22]: ▶ ! mypy estructuras_arbitrarias_1.py
```

estructuras_arbitrarias_1.py:2: error: Bracketed expression "[...]" is not valid as a type [valid-type]
estructuras_arbitrarias_1.py:2: note: Did you mean "List[...]"?
Found 1 error in 1 file (checked 1 source file)

In [23]:  %%writefile estructuras_arbitrarias_2.py

```
def suma_lista(lista: list[int]) -> int:
    return sum(lista)
```

Overwriting estructuras_arbitrarias_2.py

In [24]:  ! mypy estructuras_arbitrarias_2.py


Success: no issues found in 1 source file

In [25]:  %%writefile estructuras_arbitrarias_3.py

```
from typing import List

def suma_lista(lista: List[int]) -> int:
    return sum(lista)
```

Overwriting estructuras_arbitrarias_3.py


In [26]:  ! mypy estructuras_arbitrarias_3.py

Success: no issues found in 1 source file

In [27]:  %%writefile estructuras_arbitrarias_4.py

```
def primero(par: tuple[int, str]) -> int:
    x, _ = par
    return x
```

Overwriting estructuras_arbitrarias_4.py

In [28]:  ! mypy estructuras_arbitrarias_4.py

Success: no issues found in 1 source file

In [29]:  %%writefile estructuras_arbitrarias_5.py


```
from typing import List, Tuple, Set

TipoPerro = Tuple[str, float, Set[str]]
ListaDePerros = List[TipoPerro]

perros: ListaDePerros = \
    [("Pipo", 0.75, {"caminar", "perseguir pájaros", "caramelos"}),
     ("Blacky", 0.35, {"dormir", "morder otros perros", "sus chuches"})]

def estatura_perruna(p: TipoPerro) -> float:
    return p[1]
```

Overwriting estructuras_arbitrarias_5.py

In [30]:  ! mypy estructuras_arbitrarias_5.py

Success: no issues found in 1 source file

In [31]:  %%writefile estructuras_arbitrarias_6.py

```
from typing import List, Any

Array_22 = Any # [[int]]


def suma_diag(vector: Array_22) -> int:
    return vector[0][0] + vector[1][1]

# Más tarde podemos quizá ofrecer un tipo más específico

ArrayList_22 = List[List[int]]

def suma_diag_2(vector: ArrayList_22) -> int:
    return vector[0][0] + vector[1][1]
```

Overwriting estructuras_arbitrarias_6.py

In [32]:  ! mypy estructuras_arbitrarias_6.py

Success: no issues found in 1 source file

In [33]:  %%writefile estructuras_arbitrarias_7.py


```
import numpy as np

MiniEntero = np.int8

def doble(n: MiniEntero) -> MiniEntero:
    return MiniEntero(2*n)

a = np.int8(100)
print(type(a)) # <class 'numpy.int8'>
print(doble(a)) # -56, la explicación requiere conocer la representación
```

Overwriting estructuras_arbitrarias_7.py

In [34]:  ! mypy estructuras_arbitrarias_7.py

Success: no issues found in 1 source file

Valoración final y referencias

Las anotaciones de tipos son útiles incluso cuando no se use una herramienta de chequeo. En resumen, hacen el código más legible, para quien programa y para quienes leen el código en otro momento. Por lo tanto, se facilita también el mantenimiento de los programas.

En multitud de lugares puede leerse más información sobre las anotaciones en Python, así como las ventajas de adoptar la costumbre de anotar el código diseñado:

- <https://florimond.dev/en/posts/2018/07/why-i-started-using-python-type-annotations-and-why-you-should-too/>
- <https://towardsdatascience.com/type-annotations-in-python-d90990b172dc>
- <https://peps.python.org/pep-0484/>