



Programación. Python

Anotaciones de tipos

Declaración explícita de tipos. En C++

*Tipos del resultado
de una función*

Tipos de parámetros

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

*Tipos de constantes
o de variables*

Tipado estático

Comprobación en tiempo de compilación

Declaración explícita de tipos. En Python

*Tipos de constantes
o de variables*

Tipos de parámetros

*Tipo del resultado
de una función*

```
n: int = 37
frase: str = "El mundo era tan reciente que muchas cosas carecían de nombre"
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)
```

*Tipos de constantes
o de variables*

Tipado **dinámico**

No comprobación en tiempo de compilación

```
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'>
```

No comprobación estática de tipos en python

```
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, 1, 2, 3]
```

```
345345345345345345345345345345345345
```

```
def suma(a: int, b: str) -> int:  
    return a+b
```

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

9

TypeError Traceback (most recent call last)

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 posibles

```
n: int = 7
e: float = 2.71828
z: complex = 1-5.0j
afirmativo: bool = True
frase: str = "El lagarto 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"])
```

- Tipos de datos estándar

```
7 2.71828 (1-5j) True
El lagarto 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'>
```

Anotaciones posibles

```
def mi_fun(n: "tipo_a", m: list) -> "tipo_c":  
    return (n, n)
```

```
print(mi_fun.__annotations__)           {'n': 'tipo_a', 'm': <class 'list'>, 'return': 'tipo_c'}
```

- Nombres arbitrarios,
a título informativo

```
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)
```

```
<3, 4> <4.5, 6.0>
```

```
print(alargar.__annotations__)
```

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

- Clases definidas
por el usuario

Anotaciones posibles

```
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)      [1, 2, 3] (2, 'dos') {1, 2, 3} {'C': 34, 'A': 23}
print(__annotations__["b"])   [<class 'int'>]
print(__annotations__["par"]) (<class 'int'>, <class 'str'>)
print(__annotations__["conj"]) {<class 'int'>}
print(__annotations__["dicc"]) {<class 'str'>: <class 'int'>}
```

```
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)      [1, 2, 3] (2, 'dos') {1, 2, 3} {'C': 34, 'A': 23}
print(__annotations__["b"])    list[int]
print(__annotations__["par"])  tuple[int, str]
print(__annotations__["conj"]) {<class 'int'>}
print(__annotations__["dicc"]) dict[str, int]
```

- Estructuras de datos definidas por el usuario

*Ambas notaciones
No estándar*

- Estructuras de datos definidas por el usuario (de otro modo)

Anotaciones con typing

(1 de 2)

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

```
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]]]
```

```
p: ListaDePerros = [1, 2, 3]
print(__annotations__["perros"])     typing.List[typing.Tuple[str, float, typing.Set[str]]]
```

Ojo: los tipos siguen sin comprobarse

Anotaciones con typing (2 de 2)

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__) {'a': <class 'int'>, 'b': <class 'int'>, 'return': typing.Optional[float]}
print(divide(6, 2))           3
print(divide(6, 0))           None
```

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)) -1.0
print(solve_ec_1_grado(0, 2)) None
print(solve_ec_1_grado(0, 0)) Infinitas soluciones
```

```
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))           64
```

Comprobaciones de tipos con mypy

```
%%writefile diez_bien.py
```

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

Overwriting diez_bien.py

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

```
! mypy diez_bien.py
```

Success: no issues found in 1 source file

```
%%writefile diez_mal.py
```

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

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

```
! 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)

Comprobaciones de tipos con mypy

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

```
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)
```

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

Success: no issues found in 1 source file

```
from typing import List, Tuple, Set, Any  
  
def suma_lista(lista: List[int]) -> int:  
    return sum(lista)
```

Success: no issues found in 1 source file

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

Success: no issues found in 1 source file

```
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]
```

Success: no issues found in 1 source file

Comprobaciones de tipos con mypy

```
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]
```

Success: no issues found in 1 source file

```
import numpy as np

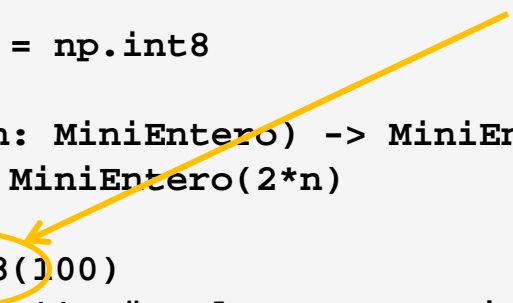
MiniEntero = np.int8

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 de los enteros en
# complemento a dos
```

Enteros en 8 bits



Success: no issues found in 1 source file



Programación. Python

Anotaciones de tipos