Type Hints in Python

PEP 484

- Python has always been a dynamically typed language, which means you don't have to specify data types
- PEP 484 introduced type hints — a way to make Python feel statically typed.
- While type hints can help structure our projects better, they are just that hints and by default do not affect runtime
- However, there is a way to force type checks on runtime

In [1]:
```
#Let's start simple by exploring a function without type hints
def add_numbers(num1, num2):
    return num1 + num2
print(add_numbers(3, 5))
```

```
8
```

Here's how you can add type hints to our function:

- Add a colon and a data type after each function parameter
- Add an arrow -> and a data type after the function to specify the return data type

```
variable name: data type = value
```

In [2]:
```
#function with type hint
def add_numbers(num1: int, num2: int) -> int:
    return num1 + num2
print(add_numbers(3, 5))
```

```
8
```

In [3]:
```
#If you're working with a function that shouldn't return anything,
#you can specify None as the return type

def add_numbers(num1: int, num2: int) -> None:
    print(num1 + num2)
add_numbers(3, 5)
```

```
8
```

In [4]:
```
#you can also set a default value for the parameter
def add_numbers(num1: int, num2: int = 10) -> int:
    return num1 + num2
print(add_numbers(3))
```

```
13
```

In [5]:
```
#what if we decide to call the add_numbers() function with floating-point numbers ? Let's check:

def add_numbers(num1: int, num2: int) -> int:
    return num1 + num2
print(add_numbers(3.5, 5.11))
```

```
8.61
```

As you can see, everything still works. Adding type hints has no runtime effect by default.

A static type checker like mypy can solve this "issue"

Let's explore variable annotations next.

Variable Annotations :

Just like with functions, you can add type hints to variables.

While helpful,type hints on variables can be a bit too verbose for simple functions & scripts.

In [6]:
```
#Let's take a look at an example:
a: int = 3
b: float = 3.14
c: str = 'abc'
d: bool = False
e: list = ['a', 'b', 'c']
f: tuple = (1, 2, 3)
g: dict = {'a': 1, 'b': 2}
```

In [7]:
```
#You can also include type annotations for variables inside a function:
def greet(name: str) -> str:
    base: str = 'Hello, '
    return f'{base}{name}'
greet('Bob')  # Hello, Bob
```

Out[7]:
```
'Hello, Bob'
```

let's explore the built-in typing module

In [8]:
```
from typing import List, Tuple, Dict
e: List[str] = ['a', 'b', 'c']
f: Tuple[int, int, int] = (1, 2, 3)
g: Dict[str, int] = {'a': 1, 'b': 2}
```

In [11]:
```
#Let's see how to include these to a function.
#It will take a list of floats and returns the same list with the items squared:

def squre(arr: List[float]) -> List[float]:
    return [x ** 2 for x in arr]
print(squre([1, 2, 3]))
```

```
[1, 4, 9]
```

let's explore one important concept in type hints – the Union operator.

It allows you to specify multiple possible data types for variables and return values.

In [10]:
```
#Here's the implementation of the previous function:
from typing import Union

def square(arr: List[Union[int, float]]) -> List[Union[int, float]]:
    return [x ** 2 for x in arr]
print(squre([1, 2, 3]))
```

```
[1, 4, 9]
```

The function can now both accept and return a list of integers or floats, warning-free.

Finally, let's explore how to force type checks on runtime.

Forcing Type Checks on Runtime

You can force type checks on runtime with mypy.

Before doing so, install it with either pip or conda:

pip install mypy

conda install mypy

© Nitheesh Reddy