Does a Python variable have a type?

In the last chapter, we saw that a variable ‘n’ can be assigned any type of object and the type() function reports the class of which ‘n’ is an instance of, based on the type of object assigned to it.

In many popular programming languages (that are compiled), a variable is declared to be of a specific type and then assigned a value that is compatible with the type of the variable. If the variable is assigned a value that is not compatible with the type of the variable, the compiler catches it and raises an error highlighting the condition.

And in Python, variables are never declared separately and it does not have a type. A variable in Python can be assigned value of any type. In effect the variable is just a name for the object that it points to

This has a few implications which a developer needs to be aware of

1. **Do not worry about declaring variables:**

In fact, you cannot declare a variable to be of a particular type in python. As a developer you just focus on the objects based on which your program logic is built and assign those objects an easily understandable name for reference.

2. **Defining types for function and method parameters**

You cannot define a function to take in a particular type of value(s) as parameters. The below code listing works to add two integer values

```
#function add
def add(n,m):
    return n + m
```

Calling this function with integer parameters 10 and 20 returns the sum of these two values

```
>>> add(10, 20)
30
```

You can also see that this simple function can work with floats as well

```
>>> add(1.1, 46)
47.1
>>> add(8, 23.5)
31.5
```

Now let us try adding two strings

```
>>> add("Hello", " World")
'Hello World'
```

And it works!!! You can see that when passing two strings, it returns the concatenated value

Let us try it on a few other types as well

```
>>> add(['one','two'], ['three','four'])
['one', 'two', 'three', 'four']
>>> add((1,2), (3,4))
(1, 2, 3, 4)
```

On lists and tuples its appends them and returns the result.

As you can see, this simple add method we have written works on a variety of object types passed in as parameter to it.

How does this work?

Trying an invalid cases shows us the secret

Now let us try to add an integer with a string

```
>>> add (1, "Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The only logic we have used in our custom add method is ‘+’ operator. All the types which has support for ‘+’ operator returned a proper result. And when the types passed in are not supported by + operation, it reports a type error.

So, it is possible to create a custom class that supports ‘+’ operation and pass the objects of this custom class to the add function as in the below listing

```
#Class that supports + operator
class magic_plus:
    #Implements the + operation
    def __add__(self, other):
        return 'Magic Added'
```

Calling add function with objects of this custom class returns the string ‘Magic Added’ which is the implementation of ‘+’ on this class

```
>>> add(magic_plus(), magic_plus())
'Magic Added'
```

Overloading

You might be well aware that overloading is having methods or functions with the same name, but with different types or no. of parameters.

We have seen in the example above that the function add could take in any type of parameter. Any function or method in Python is implicitly overloaded for the type of the parameters.

Generally, its left to the caller to pass in the correct types of objects to a function. But in case the function expects only a certain types of objects as input, care should be taken to throw back the right exceptions when an unexpected type is passed to it.

We will see later that a function can be defined to take in any no. of parameters as well, effectively making a single function in python overloaded for the type of parameters and no. of parameters passed in. We cannot define multiple functions with the same name in Python.

And now types:

Python 3 introduced type hints that made possible specifying hints of the types for the function parameters and return types. In python 3, it is possible to write the add function as below (renamed as int_add)

```
#Function that accepts and returns int
def int_add(n: int, m: int) -> int:
    return n + m
```

Note however that it's only a hint and the runtime behavior of the add function does not change in any way. We can still pass in any object that supports '+' operation to this function, including the custom class we have created as can be seen in the below listing

```
>>> int_add(10, 20)
30
>>> int_add(1.1, 46)
47.1
>>> int_add(8, 23.5)
31.5
>>> int_add("Hello", " World")
'Hello World'
>>> int_add(['one','two'], ['three','four'])
['one', 'two', 'three', 'four']
>>> int_add((1,2), (3,4))
(1, 2, 3, 4)
>>> int_add(magic_plus(), magic_plus())
'Magic Added'
```

And passing in objects that do not support '+' operations yields the same type error that we got with the add function

```
>>> int_add (1, "Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in int_add
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The type hints are only hints that can be leveraged by IDEs and third party libraries for type checking and can be used for documentation. Under the hoods the type hints are stored in the attribute __annotations__ of the function object. This information can be used by the code for type checking if needed. Python runtime does not do any automatic type checking based on the type hints.

Note that specifying type hints is optional and need not be specified if not required.

Next in our list, we will get into the concept of class in Python and explore it

* All the source files used in the samples are available under the src directory of the current path