Annotations in Python

Python is a robust and dynamically typed programming language. It has a straightforward syntax similar to writing plain English and is backed by a massive pool of libraries and features. One such feature is annotations. Annotations are arbitrary Python expressions that grant hints about the data type of variables, function parameters, and function return type.

Annotations aim to improve the readability and understanding of source code and are interpreted by third-party libraries to provide effective and time-saving services such as syntax hints, data type checking, data type hinting in IDEs or Integrated Development Environments, auto-completion of code, and automated or AI-driven documentation generation. There are two types of annotations in Python: function annotations and variable annotations. In this article, we will talk about both the annotations types in Python with the help of relevant examples.

Variable Annotations in Python

Variable annotations are annotations expressions that aim to provide details about variables' data types in Python. Variable annotations have the following syntax.

```
<variable>: <expression> = <initial value>
```

Annotations expressions are written between the variable name and its initial value, prefixed with a colon or `:`. Let us look at some examples to understand this better. Refer to the following Python code.

```
name: str = "Vaibhav"
age: int = 20
language: str = "Python"
student: bool = True
height: float = 5.9
print("name:", name)
print("age:", age)
print("language:", language)
print("student:", student)
print("height:", height)
```

Output:

```
name: Vaibhav
age: 20
language: Python
student: True
height: 5.9
```

In the above example, we use in-built Python data types for expressions. In place of them, we can also use strings and provide detailed and brief descriptions for variables. The following Python code depicts the same.

```
name: "Name of the person" = "Vaibhav"
age: "Age of the person" = 20
language: "Favorite programming language of the person" = "Python"
student: "Is the person a student?" = True
height: "Height of the person in feet" = 5.9
print("name:", name)
print("age:", age)
print("language:", language)
print("student:", student)
print("height:", height)
```

Output:

```
name: Vaibhav
age: 20
language: Python
student: True
height: 5.9
```

We can use the `__annotations__` attribute to access all the annotations. This attribute is a dictionary where keys are the variables and values are the annotations expressions. Note that this attribute will only provide details about the variables and not the functions if there are any. Refer to the following Python code for the same.

```
name: "Name of the person" = "Vaibhav"
age: "Age of the person" = 20
language: "Favorite programming language of the person" = "Python"
student: "Is the person a student?" = True
height: "Height of the person in feet" = 5.9
print(__annotations__)
```

Output:

```
{
    'name': 'Name of the person',
    'age': 'Age of the person',
    'language': 'Favorite programming language of the person',
    'student': 'Is the person a student?',
    'height': 'Height of the person in feet'
}
```

For the first example, the output will be as follows.

```
{
    'name': <class 'str'>,
    'age': <class 'int'>,
    'language': <class 'str'>,
    'student': <class 'bool'>,
    'height': <class 'float'>
}
```

So far, we have only discussed primitive data types such as `int`, `float`, and `str`. Now let us understand how to write annotation expressions for complex data types such as `list`, `tuple`, `set`, `list` of objects, etc. For this, we will use the `typing` module. The `typing` module is a part of Python's standard library. Let us understand how to use it for complex data types with the help of an example. Refer to the following Python code for the same.

```
from typing import List, Tuple, Set

def user():
    return {
        "name": "Vaibhav",
        "username": "vaibhav",
        "password": "vaibhav"
    }

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

middlewares: List[str] = []
points: Tuple[Point] = tuple([Point(0, 0), Point(1, 1)])
numbers: Set[int] = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
users: List[dict] = [user()]
utils: List["function"] = [sum, len, sorted]
pairs: List[List[int]] = [[1, 2], [2, 3], [3, 4]]
print("middlewares:", middlewares, end = "\n\n")
print("points:", points, end = "\n\n")
print("numbers:", numbers, end = "\n\n")
print("utils:", utils, end = "\n\n")
print("users:", users, end = "\n\n")
print("pairs:", pairs, end = "\n\n")
print(__annotations__)
```

Output:

```
middlewares: []
```

```
points: (<__main__.Point object at 0x7fc658e454c0>,
<__main__.Point object at 0x7fc658cef610>)

numbers: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

utils: [<built-in function sum>, <built-in function len>, <built-in function
sorted>]

users: [{'name': 'Vaibhav', 'username': 'vaibhav', 'password': 'vaibhav'}]

pairs: [[1, 2], [2, 3], [3, 4]]

{
    'middlewares': typing.List[str],
    'points': typing.Tuple[__main__.Point],
    'numbers': typing.Set[int],
    'users': typing.List[dict],
    'utils': typing.List[ForwardRef('function')],
    'pairs': typing.List[typing.List[int]]
}
```

The `typing` module has classes `List`, `Tuple`, and `Set` for `list`, `tuple`, and `set`, respectively, that act as their generic versions. Apart from these three, there are other generic classes as well, such as `Dict`, `FrozenSet`, `DefaultDict`, and `OrderedDict`.

These generic classes can be used to provide annotations expressions for variables. Next to these classes, inside `[]` brackets, primitive data types, string descriptions, classes, or other generic classes from the same module are placed. Note that they can be used to provide expressions for functions as well, about which we will learn in a bit. To learn about the `typing` module, refer to the official documentation here.

Function Annotations in Python

Function annotations are annotations expressions that aim to provide details about functional parameters data types and functions' return values data types in Python. Function annotations have the following syntax.

```
def function(
    <para>: <expression>, <para>: <expression> = <default value>
) -> <expression>:
```

Annotation expressions for parameters are placed next to parameters separated by a colon or `:`. In case there is any default value, they are placed after annotation expressions. For functions' returns types, the function signature is followed by an `->` or arrow and the annotation expression. Note that the colon is placed at the very end.

Let us understand function annotations with the help of some relevant examples. Refer to the following Python code for the same.

```
```

```python
from typing import List, Tuple

def create_user(name: str, age: int, hobbies: List[str] = []) -> dict:
    return {
        "name": name,
        "age": age,
        "hobbies": hobbies
    }

def create_users(users: List[Tuple]) -> List[dict]:
    result = []

    for user in users:
        result.append(
            create_user(name = user[0], age = user[1], hobbies = user[2])
        )

    return result

u1: dict = create_user("Vaibhav", 20, ["Football", "Video Games"])
data = [
    ("Rick", 40, ["Shooting"]),
    ("Derly", 38, ["Archery", "Tracking"]),
    ("Maggie", 25, []),
    ("Carol", 32, ["Cooking"]),
]
users: List[dict] = create_users(data)
print(u1)
print(users)
print(__annotations__)
```

Output:

```
{
    'name': 'Vaibhav',
    'age': 20,
    'hobbies': ['Football', 'Video Games']
}
[
    {
        'name': 'Rick',
        'age': 40,
        'hobbies': ['Shooting']
    }, {
        'name': 'Derly',
        'age': 38,
        'hobbies': ['Archery', 'Tracking']
    }, {
        'name': 'Maggie',
        'age': 25,
        'hobbies': []
```

```
    }, {
        'name': 'Carol',
        'age': 32,
        'hobbies': ['Cooking']
    }
]
{
    'u1': <class 'dict'>,
    'users': typing.List[dict]
}
```

As we can see, the `create_user()` function accepts three values, namely, `name`, `age`, and `hobbies`, and returns a dictionary or `dict`. The `create_users()` method accepts a list of tuples representing a list of users. This method returns a list of dictionaries. The result of the method call to the `create_user()` method is stored in a variable `u1`, which is of type `dict`. And, the result of the function call to the `create_users()` method is stored in a variable `users`, which is of type `List[dict]`. The `__annotations__` attribute will only deliver details about the variables. To fetch annotation details about the functions, we can use the `__annotations__` attribute of functions. The following Python code depicts the same.

```
from typing import List, Tuple

def create_user(name: str, age: int, hobbies: List[str] = []) -> dict:
    return {
        "name": name,
        "age": age,
        "hobbies": hobbies
    }

def create_users(users: List[Tuple]) -> List[dict]:
    result = []

    for user in users:
        result.append(
            create_user(name = user[0], age = user[1], hobbies = user[2])
        )

    return result

print(create_user.__annotations__)
print(create_users.__annotations__)
```

Output:

```
{
    'name': <class 'str'>,
    'age': <class 'int'>,
    'hobbies': typing.List[str],
    'return': <class 'dict'>
```

```
}
{
    'users': typing.List[typing.Tuple],
    'return': typing.List[dict]
}
```

The output dictionary will include all the annotation details. Note that for the return type, `"return"` is the key in the dictionary. For parameters, parameter names are the keys.