Python - Anotações de Função

Anotações de função

O recurso de anotação de função do Python permite adicionar metadados explicativos adicionais sobre os argumentos declarados em uma definição de função e também o tipo de dados de retorno.

Embora você possa usar o recurso docstring do Python para documentação de uma função, ele pode se tornar obsoleto se forem feitas certas alterações no protótipo da função. Conseqüentemente, o recurso de anotação foi introduzido em Python como resultado do PEP 3107.

As anotações não são consideradas pelo interpretador Python durante a execução da função . Eles são principalmente para IDEs Python para fornecer documentação detalhada ao programador.

Anotações são quaisquer expressões Python válidas adicionadas aos argumentos ou retornam o tipo de dados. O exemplo mais simples de anotação é prescrever o tipo de dados dos argumentos. A anotação é mencionada como uma expressão após colocar dois pontos na frente do argumento.

Exemplo

```
def myfunction(a: int, b: int):
   c = a+b
   return c
```

Lembre-se de que Python é uma linguagem de tipo dinâmico e não impõe nenhuma verificação de tipo em tempo de execução. Portanto, anotar os argumentos com tipos de dados não tem nenhum efeito ao chamar a função. Mesmo que sejam fornecidos argumentos não inteiros, o Python não detecta nenhum erro.

```
def myfunction(a: int, b: int):
   c = a+b
   return c

print (myfunction(10,20))
print (myfunction("Hello ", "Python"))
```

Ele produzirá a seguinte **saída** -

```
30
Hello Python
```

Anotações de função com tipo de retorno

As anotações são ignoradas em tempo de execução, mas são úteis para IDEs e bibliotecas de verificação de tipo estático, como mypy.

Você também pode fornecer anotações para o tipo de dados de retorno. Após os parênteses e antes do símbolo de dois pontos, coloque uma seta (->) seguida da anotação. Por exemplo -

Exemplo

```
def myfunction(a: int, b: int) -> int:
   c = a+b
   return c
```

Anotações de função com expressão

Como o uso do tipo de dados como anotação é ignorado em tempo de execução, você pode colocar qualquer expressão que atue como metadados para os argumentos. Portanto, a função pode ter qualquer expressão arbitrária como anotação, como no exemplo a seguir -

Exemplo

```
def total(x : 'marks in Physics', y: 'marks in chemistry'):
   return x+y
```

Anotações de função com argumentos padrão

Se quiser especificar um argumento padrão junto com a anotação, você precisará colocá-lo após a expressão da anotação. Os argumentos padrão devem vir depois dos argumentos obrigatórios na lista de argumentos.

Exemplo

```
def myfunction(a: "physics", b:"Maths" = 20) -> int:
   c = a+b
   return c
print (myfunction(10))
```

A função em Python também é um object e um de seus atributos é __annotations__. Você pode verificar com a função dir().

```
print (dir(myfunction))
```

Isso imprimirá a lista do objeto myfunction contendo __annotations__ como um dos atributos.

```
['__annotations__', '__builtins__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
```

O próprio atributo __annotations__ é um dicionário no qual os argumentos são chaves e as anotações são seus valores.

```
def myfunction(a: "physics", b:"Maths" = 20) -> int:
   c = a+b
   return c
print (myfunction.__annotations__)
```

Ele produzirá a seguinte **saída** -

```
{'a': 'physics', 'b': 'Maths', 'return': <class 'int'>}
```

Você pode ter argumentos posicionais arbitrários e/ou palavras-chave arbitrárias para uma função. Anotações também podem ser fornecidas para eles.

```
def myfunction(*args: "arbitrary args", **kwargs: "arbitrary keyword args") -> int:
   pass
print (myfunction.__annotations__)
```

Ele produzirá a seguinte **saída** -

```
{'args': 'arbitrary args', 'kwargs': 'arbitrary keyword args', 'return': <class 'int'>}
```

Caso você precise fornecer mais de uma expressão de anotação para um argumento de função, forneça-a na forma de um objeto de dicionário na frente do próprio argumento.

```
def division(num: dict(type=float, msg='numerator'), den: dict(type=float, msg='denom
   return num/den
print (division.__annotations__)
```

Ele produzirá a seguinte **saída** -

{'num': {'type': <class 'float'>, 'msg': 'numerator'}, 'den': {'type': <class 'float'>, 'msg': 'denomir