

# 注解annotation

Python是动态语言，变量可以随时被赋值并改变类型，也就是说Python的变量是运行时决定的。

```
1 def add(x, y):
2     return x + y
3
4 print(add(4, 5))
5 print(add('mag', 'edu'))
6 print(add([10], [11]))
7
8 print(add(4, 'abc')) # 不到运行时，无法判断类型是否正确
```

动态语言缺点：

1. 难发现：由于不能做任何类型检查，往往到了运行时问题才显现出来，或到了线上运行时才暴露出来
2. 难使用：函数使用者看到函数时，并不知道函数设计者的意图，如果没有详尽的文档，使用者只能猜测数据的类型。对于一个新的API，使用者往往不知道该怎么使用，对于返回的类型更是不知道该怎么使用

动态类型对类型的约束不强，在小规模开发的危害不大，但是随着Python的广泛使用，这种缺点确实对大项目的开发危害非常大。

如何解决这种问题呢？

1. 文档字符串。对函数、类、模块使用详尽的使用描述、举例，让使用者使用帮助就能知道使用方式。但是，大多数项目管理不严格，可能文档不全，或者项目发生变动后，文档没有更新等等。
2. 类型注解：函数注解、变量注解

## 函数注解

```
1 def add(x:int, y:int) -> int:
2     """
3
4     :param x: int
5     :param y: int
6     :return: int
7     """
8     return x + y
9
10 help(add)
11 print(add(4, 5))
12 print(add('mag', 'edu'))
```

函数注解

- 3.5版本引入
- 对函数的形参和返回值的类型说明
- 只是对函数形参类型、返回值类型做的辅助的说明，非强制类型约束
- 第三方工具，例如Pycharm就可以根据类型注解进行代码分析，发现隐藏的Bug
- 函数注解存放在函数的属性 `__annotations__` 中，字典类型

```
1 {'x': <class 'int'>, 'y': <class 'int'>, 'return': <class 'int'>}
```

## 类型注解

```
1 i:int = 3
2 j:str = 'abc'
3 k:str = 300
4 print(i, j, k)
```

### 类型注解

- 3.6版本引入
- 对变量类型的说明，非强制约束
- 第三方工具可以进行类型分析和推断

## 类型检查

函数传参经常传错，如何检查？

可以在函数内部写isinstance来判断参数类型是否正确，但是检查可以看做不是业务代码，写在里面就是侵入式代码。那如何更加灵活的检查呢？

- 非侵入式代码
- 动态获取待检查的函数的参数类型注解
- 当函数调用传入实参时，和类型注解比对

能否使用函数的 `__annotations__` 属性吗？虽然Python 3.6之后，字典记录了录入序，但是我们还是要认为字段是无顺序的。那如何和按位置传实参对应呢？

使用inspect模块

## inspect模块

inspect模块可以获取Python中各种对象信息。

- `inspect.isfunction(add)`，是否是函数
- `inspect.ismethod(pathlib.Path().absolute)`，是否是类的方法，要绑定
- `inspect.isgenerator(add)`，是否是生成器对象
- `inspect.isgeneratorfunction(add)`，是否是生成器函数
- `inspect.isclass(add)`，是否是类
- `inspect.ismodule(inspect)`，是否是模块
- `inspect.isbuiltin(print)`，是否是内建对象

还有很多is函数，需要的时候查阅inspect模块帮助

`inspect.signature(callable, *, follow_wrapped=True)`

- 获取可调用对象的签名
- 3.5增加follow\_wrapped，如果使用functools的wraps或update\_wrapper，follow\_wrapped为True跟进被包装函数的 `__wrapped__`，也就是去获取真正的被包装函数的签名

```
1 import inspect
2
```

```

3 def add(x:int, /, y:int=5, *args, m=6, n, **kwargs) -> int:
4     return x + y + m + n
5
6 sig = inspect.signature(add) # 获取签名
7 print(sig)
8 print(sig.return_annotation) # 返回值注解
9 params = sig.parameters # 所有参数
10 print(type(params))
11 print(params) # 有序字典OrderedDict
12
13 for k,v in params.items():
14     print(type(k), k, type(v), v, sep='\t')

```

inspect.Parameter

- 4个属性
  - name, 参数名, 字符串
  - default, 缺省值
  - annotation, 类型注解
  - kind, 类型
    - POSITIONAL\_ONLY, 只接受仅位置传参
    - POSITIONAL\_OR\_KEYWORD, 可以接受关键字或者位置传参
    - VAR\_POSITIONAL, 可变位置参数, 对应\*args
    - KEYWORD\_ONLY, 对应\*或者\*args之后的出现的非可变关键字形参, 只接受关键字传参
    - VAR\_KEYWORD, 可变关键字参数, 对应\*\*kwargs
- empty, 特殊类, 用来标记default属性或者annotation属性为空

```

1 import inspect
2
3 def add(x:int, /, y:int=5, *args, m=6, n, **kwargs) -> int:
4     return x + y + m + n
5
6 sig = inspect.signature(add) # 获取签名
7 print(sig)
8 print(sig.return_annotation) # 返回值注解
9 params = sig.parameters # 所有参数
10 print(type(params))
11 print(params) # 有序字典OrderedDict
12
13 for k,v in params.items():
14     print(type(k), k)
15     t:inspect.Parameter = v # 这一步是多余的, 但是t使用了变量注解
16     print(t.name, t.default, t.kind, t.annotation, sep='\t')
17     print('-' * 30)

```

## 参数类型检查

有如下函数

```
1 def add(x, y:int=7) -> int:
2     return x + y
3
4 add(4, 5)
5 add('mag', 'edu')
```

请检查用户的输入是否符合参数类型注解的要求

分析：

- 调用时，用户才会传入实参，才能判断实参是否符合类型要求
- 调用时，让用户感觉上还是调用原函数
- 如果类型不符，提示用户

先实现对add函数的参数类型的提取

```
1 import inspect
2
3 def add(x, y:int=7) -> int:
4     return x + y
5
6 def check(fn):
7     sig = inspect.signature(fn)
8     params = sig.parameters
9
10    for k, v in params.items():
11        print(k, v.default, v.annotation)
12
13 check(add)
14 add(4, 5) # 如何对它进行判断
15
16 输出结果
17 x <class 'inspect._empty'> <class 'inspect._empty'>
18 y 7 <class 'int'>
```

如何解决add(4, 5)调用问题？

```
1 import inspect
2 from functools import wraps
3
4 def check(fn):
5     @wraps(fn)
6     def wrapper(*args, **kwargs):
7         sig = inspect.signature(fn)
8         params = sig.parameters
9
10        print(args, kwargs)
11        print(params)
12        ret = fn(*args, **kwargs)
```

```

13         return ret
14     return wrapper
15
16 @check
17 def add(x, y:int=7) -> int:
18     return x + y
19
20 add(4, 5)

```

对于按位置传参如何解决？

```

1  import inspect
2  from functools import wraps
3
4  def check(fn):
5      @wraps(fn)
6      def wrapper(*args, **kwargs):
7          sig = inspect.signature(fn)
8          params = sig.parameters
9
10         print(args, kwargs)
11         print(params)
12
13         values = tuple(params.values())
14         for i, v in enumerate(args): # 迭代谁?
15             if values[i].annotation is not values[i].empty and isinstance(v,
values[i].annotation):
16                 print('{ }={ } is ok'.format(values[i].name, v)) # 有类型注解的
检查
17
18         ret = fn(*args, **kwargs)
19         return ret
20     return wrapper
21
22 @check
23 def add(x, y:int=7) -> int:
24     return x + y
25
26 add(4, 5)

```

如果按照关键字传参如何解决？

```

1  import inspect
2  from functools import wraps
3
4  def check(fn):
5      @wraps(fn)
6      def wrapper(*args, **kwargs):
7          sig = inspect.signature(fn)
8          params = sig.parameters
9
10         print(args, kwargs)
11         print(params)
12
13         values = tuple(params.values())
14         for i,v in enumerate(args): # 迭代谁?

```

```
15         if values[i].annotation is not values[i].empty and isinstance(v,
values[i].annotation):
16             print('{}={} is ok'.format(values[i].name, v)) # 有类型注解的
检查
17
18         for k,v in kwargs.items(): # 迭代谁?
19             if params[k].annotation is not inspect._empty and isinstance(v,
params[k].annotation):
20                 print('{}={} is ok'.format(k, v)) # 有类型注解的检查
21
22         ret = fn(*args, **kwargs)
23         return ret
24     return wrapper
25
26 @check
27 def add(x, y:int=7) -> int:
28     return x + y
29
30 add(x=4, y=5)
```