Python 代码结构

函数

名称到值的绑定提供了有限的抽象手段，通过函数，我们获得了一个更强大的抽象技巧——名称通过它可以绑定到复合操作上，并可以作为一个单元来引用。函数可以接受任何类型的输入作为参数，并返回任何类型的结果。

使用函数可以做以下两件事情：

- 定义函数
- 调用函数

函数定义包含`def`语句，它表明了函数名称`<func_name>`和一系列带有名字的形式参数`<formal parameters>`（可选）。之后的语句叫做函数体，`return`（返回）指定了函数的返回表达式`<return expression>`。

```
def <func_name>(<formal parameters>):
    return <return expression>
```

第二行必须缩进！返回表达式并不是立即求值，它储存为新定义函数的一部分，并且只在函数最终调用时会被求出。

```
def do_noting():
    pass
do_noting
```

Python函数命名规范和变量命名一样（必须使用字母或者下划线`_`开头，仅能含有字母、数字和下划线）。上面代码示例中的`pass`表明函数没有做任何事情，仅用来进行占位。

```
def square(x):
    return x * x
```

定义了函数之后，我们使用调用表达式来调用它：

```
square(21)  # 441
square(2 + 5)  # 49
square(square(3))  # 81
```

一个函数可以接受任何数量（包括0）的任何类型的值作为输入变量，并且返回任何数量（包括0）的任何类型的结果。如果函数不显示调用`return`，那么会默认返回`None`。

注意：关于`None`和`False`的区别，参考教材p.78。

函数的局部变量

函数体的执行会为这个函数的局部变量引入一个新的**命名空间**。所有函数体中的赋值语句（assignment statement），都会把变量名和值存在这个命名空间中。

函数体中引用一个变量时，首先查看这个函数的命名空间，如果这个函数定义包裹在其它函数定义中，就依次查看外围函数的命名空间，然后查看**全局**命名空间（也就是函数所属的module的命名空间），最后查看Python的内置类型和变量的命名空间。

函数的形参也是存在于函数的局部命名空间中。函数的局部命名空间会在每次调用和返回时进行创建初始化和删除。

函数调用的实参传递是通过赋值语句做的，所以传递的是对象的引用。对于类似序列的可变（mutable）类型，如果其作为参数按引用传递给函数，在函数体中改变它的值也会影响它在外围命名空间的值，就像C++中的引用。

```
alist = ['a', 'b']
def foo(list_obj):
    list_obj.append('c')
foo(alist)
print(alist)

count = 1000
print('count:', id(count))
def incr(n):
    print('old n:', id(n))
    n += 1
    print('new n:', id(n))
incr(count)
print('count again:', id(count))
count
```

函数的参数

函数在声明参数时大概有下面 4 种形式：

1. 位置参数：`def func(a, b): pass`
2. 关键字参数：`def func(a, b=1): pass`
3. 任意位置参数：`def func(a, b=1, *c): pass`
4. 任意关键字参数：`def func(a, b=1, *c, **d): pass`

位置参数（无默认值参数）

按照顺序将传入参数的值依次复制进去。

```
def foo(arg1, arg2):
    return arg1, arg2

foo('a', 'b')
```

使用位置参数必须熟记每个位置参数的含义。

关键字参数（有默认值参数）

调用函数时可以指定对应参数的名字，可以采用于函数定义不同的顺序调用。

```
def foo(arg1='a', arg2='b'):
    return arg1, arg2

foo(arg2='c', arg1='d')
foo(1, 2)
foo(1, arg1='a')  # TypeError
```

当位置参数和关键字参数同时存在时，关键字参数必须放到位置参数的后面。

```
def foo(arg1='a', arg2):  # SyntaxError
    return arg1, arg2
```

如果同时使用位置参数和关键字参数两种方式调用函数，关键字参数也必须放到位置参数之后。

调用函数时如果没有提供关键字参数的参数值时，将使用函数定义时的默认参数值。

```
foo(1)
```

如果调用函数时提供关键字参数的参数值，则将代替默认值。

注意：函数的关键字参数的参数值在函数定义时已经计算出来了，而不是在函数运行时。

```
num = 1
def bar(arg1=num):
    print(arg1)
num = 2
bar()
```

所以，在定义函数时，不要把可变的数据类型（列表、字典）当作关键字参数的参数值。函数的关键字参数只会被求值一次，不管函数被怎么调用，当关键字参数的参数值是可变对象时，在函数体中如果其值被改变，再次调用函数时其默认值就是改变后的值。

```
xyz_list = ['x', 'y', 'z']
def append_xyz(alist, blist=xyz_list):
    alist.extend(blist)
    return alist
append_xyz(['a'])
xyz_list.insert(0, 'w')
append_xyz(['a'])
```

```
def bar(n, alist=[]):
    alist.append(n)
    return alist

print(bar(1))
print(bar(2))
print(bar(3))
```

如何避免这种情况?

```
def bar(n, alist=None):
    if alist is None:
        alist = []
    alist.append(n)
    return alist

print(bar(1))
print(bar(2))
print(bar(3))
```

`None`是个内置常量，当然不能被改变，每次函数`bar()`被调用就会用这个值给`alist`赋值。

任意位置参数

任意位置参数可以接受任意数量的位置参数。当参数被用在函数内部时，`*`将一组可变数量的位置参数集合成参数值的元组。

```
def concat(*lst, sep='|'):
    return sep.join((str(i) for i in lst))
print(concat('G', 30, '@', 'Hz', sep=''))  # G30@Hz
```

注意：上面的关键词参数必须明确指明，不能通过位置推断。

```
print(concat('G', 30, '-'))  # G|30|-, Not G-30
```

通过这种方式给函数传入的所有位置参数都会以元组的形式返回输出：

```
def print_args(*args):
    print(args)

print_args(3, 2, 1, 'a', 'b', ['c', 'd'])
```

这对于编写接受可变数量的参数的函数非常有用。如果函数同时有限定的位置参数，那么`*args`会收集剩下的参数。

```
def print_more(arg1, arg2, *rest_args):
    print(arg1)
    print(arg2)
    print(rest_args)

print_more(3, 2, 1, 'a', 'b', ['c', 'd'])
```

任意关键字参数

使用`**`可以将参数收集到一个字典中，参数的名字是字典的键，对应参数的值是字典的值。

```
def print_kwargs(**kwargs):
    print(kwargs)

print_kwargs(arg1=1, arg2=2, arg3='a')
```

```
def dconcat(sep=':', **kwargs):
    for k in kwargs.keys():
        print('{}{}{}'.format(k, sep, kwargs[k]))

dconcat(hello='world', python='rocks', sep='~')
```

如果想把带有`*args`和`**kwargs`的位置参数混合起来，则它们必须按照顺序出现。

解包（Unpacking）

如果有一个列表或一个字典，我们可以把它直接作为参数传给一个函数，里面的值可以解包出来传给一个个参数。

解包为位置实参

可以把一个列表或元组解包，对应的值作为位置参数传递，调用的时候要以`*args`的形式：

```
lst = [0, 1, 2, 3]
print(*lst)  # 注意调用语法为`*args`形式

print(*range(5))
```

解包为关键字实参

将字典解包为关键字实参。字典的键作为形参的名字，是字符串，对应键的值为传递的实参。语法上调用的时候以`**kwargs`的形式。

```
def f(a, b, c):
    print("a =", a, "b =", b, "c =", c)

d = {"a":5, "c":8, "b":2}
f(**d)
```

另外，Python3.5添加的新特性（PEP 448），使得`*args`、`**kwargs`可以在函数参数之外使用：

```
a = *range(3), # 这里的逗号不能漏掉
print(a)

d = {"hello": "world", "python": "rocks"}
print({**d}["python"])
```

所谓的解包（Unpacking）实际上可以看做是去掉`()`的元组或者是去掉`{}`的字典。这一语法也提供了一个更加 Pythonic 地合并字典的方法：

```
user = {'name': "Google", 'website': "https://www.google.com"}
defaults = {'name': "Anonymous User", 'page_name': "Profile Page"}

# 合并字典的3中方法
print({**defaults, **user})
defaults.update(user)
{k:v for d in [user, defaults] for k, v in d.items()}
```

函数文档字符串

在函数体开始的部分附上函数定义说明的文档或加上详细的规范说明。一般建议先用一句话简明扼要的说明函数的功能或作用，然后对函数的参数和返回值进行具体解释，比如参数和返回值的类型及相应的意义，还可以添加函数的用法或者异常处理等内容。

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

调用Python的`help()`函数可以打印输出一个函数的文档字符串，得到其参数列表和规范的文档。

```
help(fetch_bigtable_rows)
```

```
help(print)
```

如果仅仅想得到文档字符串，可以直接查看函数的`__doc__`属性。

```
print(fetch_bigtable_rows.__doc__)
```

第一类对象（first-class object）：函数

指可以在执行期创建并作为参数传递给其他函数或传入一个变数的实体。一般第一类对象具有一下特征：

1. 可以被存入变量或其他结构
2. 可以被作为参数传递给其他方法/函数
3. 可以被作为方法/函数的返回值
4. 可以在执行期被创建，而无需在设计期全部写出
5. 有固定身份

“固有身份”是指实体有内部表示，而不是根据名字来识别，比如匿名函数，还可以通过赋值叫任何名字。在Python中，函数/方法都是第一类对象，这对于对函数式编程语言来说是必须的。所以，Python是多范式编程语言。

```
def sum_args(*args):
    return sum(args)

def run_with_positional_args(func, *args):
    return func(*args)


type(run_with_positional_args)
run_with_positional_args(sum_args, 1, 2, 3)
```

可以把函数作为列表、元组、集合和字典的元素。另外，由于函数名也是不可变的，可以把函数名用作字典的键。

内部函数

在Python中，可以在函数中定义另外一个函数：

```
def make_adder(n):
    def adder(x):
        return x + n
    return adder

add1 = make_adder(1)
add1(2)
```

另外，当需要在函数内部多次执行复杂的任务时，为了避免循环和代码的堆叠重复，也可以使用内部函数。

闭包（closure）

上面的例子中`make_adder()`函数返回的内部函数可以看作是一个闭包。闭包是一个可以由另外一个函数动态生成的函数，并且可以改变和存储函数外创建的变量的值。

```
add1  # <function __main__.make_adder.<locals>.adder>
```

```
add2 = make_adder(2)
add2(2)
```

lambda函数（匿名函数）

如果需要一个函数，但又不想费神地去命名它的时候，可以使用lambda函数（匿名函数，也是闭包）。

```
sq = lambda x: x * x
sq(2)
```

```
(lambda x: x * x)(2)
```

lambda函数接受一个或多个参数（使用逗号分隔），冒号之后的部分为函数的定义。

Python的lambda限制多多，最严重的当属它只能由一条表达式组成。这个限制主要是为了防止滥用，因为当人们发觉lambda很方便，就比较容易滥用，可是用多了会让程序看起来不那么清晰，毕竟每个人对于抽象层级的忍耐/理解程度都有所不同。

装饰器（decorator）

装饰器实质上是一个函数。它把函数作为一个输入并返回另外一个包装（修改）了输入函数之后的函数。

```
import time


def timethis(func):
    """
    Decorator that reports the execution time of the func.
    """
    def wrapper(*args, **kwargs):
        start = time.time()
        func_result = func(*args, **kwargs)
        end = time.time()
        print('function execution time:', end-start)
        return func_result
    return wrapper

def countdown(n):
    while n > 0:
        n -= 1

timethis(countdown)(1000000)
```

Python实现了专门的装饰器语法，所以我们可以使用下面的方式来“装饰”一个函数：

```
@timethis
def countdown(n):
    while n > 0:
        n -= 1

countdown(100000)
```

需要强调的是，装饰器并不会修改原始函数的参数以及返回值。使用`*args`和`**kwargs`的目的就是确保任何参数都能适用。而返回结果基本都是调用原始函数`func(*args, **kwargs)`的返回结果，其中`func`就是原始函数。

带参数的装饰器

下面的代码展示了一个Web应用中使用装饰器对用户进行权限检查的例子，当前用户（`current_user`）如果是管理员的话则提醒其无法访问并进行页面重定向，使其跳转到一个指定的页面（通过装饰器函数的参数`endpoint`和任意关键字参数`**values`来控制指定跳转的页面）。

```
def staff_redirect(endpoint, **values):
    def decorator(func):
        def decorated_function(*args, **kwargs):
            if current_user.is_staff():   # 检查用户是否为管理员
                flash('抱歉，管理用户无法访问。')
                return redirect(url_for(endpoint, **values))
            return func(*args, **kwargs)
        return decorated_function
    return decorator


@staff_redirect('management_page')
@login_required
def normal_user_page():
    ...
```

上面的代码同时展示了对一个函数使用多个装饰器的例子，其中靠近函数定义的装饰器最先执行，然后依次执行上面的。针对某一函数使用多个装饰器的时候最好清楚装饰器的作用及相互之间的关系，从而确定执行顺序。

命名空间和作用域

上面我们讲到每一个函数都拥有自己的命名空间。每个程序的主要部分定义了全局命名空间，在全局命名空间的变量是全局变量。你可以在一个函数内得到某个全局变量的值并使用它，但如果在函数内部直接对全局变量进行赋值或尝试修改它会引发`UnboundLocalError`异常，因为在函数的命名空间中并不存在这个变量。为了读取全局变量而不是函数中的局部变量，需要在变量前面显示地加关键字`global`。

```
animal = 'fruitbat'
def change_and_print_global():
    animal = 'wombat'
    print(animal)

change_and_print_global()
print(animal)
```

Python提供两个用于获取命名空间内容的函数：

- `locals()` 返回一个局部命名空间内容的字典；
- `globals()` 返回一个全局命名空间内容的字典。

```
animal = 'fruitbat'
def change_local():
    animal = 'wombat'
    print(locals())
change_local()
print(globals())
```

以两个下划线`__`开头和结尾的名称在Python中有特殊用法，比如一个函数的名称是系统变量`function.__name__`，它的文档字符串是`function.__doc__`。所以，在自定义的变量中最好不要使用`__`。

当你打开一个`.py`文件时，经常会在代码的最下面看到`if __name__ == '__main__':`，这里`__name__`是模块（module）的一个内置属性。如果`import`一个模块，那么模块`__name__`的值通常为模块文件名，不带路径或者文件拓展名。但是你也可以像一个标准程序那样直接运行模块，这时候`__name__`的值将是一个特别的缺省名`__main__`。我们可以通过`if __name__ == '__main__'`来判断是否是在直接运行该`.py`文件。我们会在这个判断里面编写直接运行此模块的代码而不用担心此模块被`import`时这些直接运行的代码也被执行。

```
#!/usr/bin/env python3.6

"""
hello_world.py
"""


def main():
    print('hello, world!')


if __name__ = '__main__':
    main()
```