关于参数（上）

之前就提到过，从结构上来看，每个函数都是一个完整的程序，因为一个程序，核心构成部分就是输入、处理、输出：

> - 它可以有**输入** —— 即，它能接收外部通过参数传递的值；
> - 它可以有**处理** —— 即，内部有能够完成某一特定任务的代码；尤其是，它可以根据 “输入” 得到 “输出”；
> - 它可以有**输出** —— 即，它能向外部输送返回值.....

所以，在我看来，有了一点基础知识之后，最早应该学习的是 “如何写函数” —— 这个起点会更好一些。

这一章的内容，看起来会感觉与Part1 F4 函数那一章部分重合。但这两章的出发点不一样：

> - Part1.E.4 函数那一章，只是为了让读者有 “阅读” 函数说明文档的能力；
> - 这一章，是为了让读者能够开始动手写函数给自己或别人用.....

为函数取名

哪怕一个函数内部什么都不干，它也得有个名字，然后名字后面要加上圆括号 `()`，以明示它是个函数，而不是某个变量。

定义一个函数的关键字是 `def`，以下代码定义了一个什么都不干的函数：

```
def do_nothing():
    pass

do_nothing()
```

为函数取名（为变量取名也一样）有些基本的注意事项：

> - 首先，名称不能以数字开头。能用在名称开头的有，大小写字母和下划线 `_`；
> - 其次，名称中不能有空格，要么使用下划线连接词汇，如，`do_nothing`，要么使用 Camel Case，如 `doNothing` —— 更推荐使用下划线；
> - 再次，名称不能与关键字重合 —— 以下是 Python 的 Keyword List：

-	Python	Keyword	List	-
and	as	assert	async	await
continue	def	del	elif	else
finally	for	from	global	if
is	lambda	None	nonlocal	not
raise	return	True	try	while

-	Python	Keyword	List	-
and	as	assert	async	await

你随时可以用以下代码查询关键字列表：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import keyword
keyword.kwlist                # 列出所有关键字
keyword.iskeyword('if')       # 查询某个词是不是关键字
```

```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
```

```
 'yield']
True
```

关于更多为函数、变量取名所需要的注意事项，请参阅：

> - PEP 8 -- Style Guide for Python Code: Naming Conventions
> - PEP 526 -- Syntax for Variable Annotations
>
> 注：PEPs，是 Python Enhancement Proposals 的缩写：https://www.python.org/dev/peps/

不接收任何参数的函数

在定义函数的时候，可以定义成不接收任何参数；但调用函数的时候，依然需要写上函数名后面的圆括号 `()`：

```
def do_something():
    print('This is a hello message from do_something().')

do_something()
```

```
This is a hello message from do_something().
```

没有 return 语句的函数

函数内部，不一定非要有 `return` 语句 —— 上面 `do_somthing()` 函数就没有 `return` 语句。但如果函数内部并未定义返回值，那么，该函数的返回值是 `None`，当 `None` 被当作布尔值对待的时候，相当于是 `False`。

这样的设定，使得函数调用总是可以在条件语句中被当作判断依据：

```
def do_something():
    print('This is a hello message from do_something().')

if not do_something():                  # 由于该函数名称的缘故，这一句代码的可读性很差……
    print("The return value of 'do_something()' is None.")
```

```
This is a hello message from do_something().
The return value of 'do_something()' is None.
```

`if not do_something():` 翻译成自然语言，应该是，“如果 `do_something()` 的返回值是‘非真’，那么：……”

接收外部传递进来的值

让我们写个判断闰年年份的函数，取名为 `is_leap()`，它接收一个年份为参数，若是闰年，则返回 `True`，否则返回 `False`。

根据闰年的定义：

> - 年份应该是 4 的倍数；
> - 年份能被 100 整除但不能被 400 整除的，不是闰年。

所以，相当于要在能被 4 整除的年份中，排除那些能被 100 整除却不能被 400 整除的年份。

```
def is_leap(year):
    leap = False
    if year % 4 == 0:
        leap = True
        if year % 100 == 0 and year % 400 != 0:
            leap = False
    return leap


is_leap(7)
is_leap(12)
is_leap(100)
is_leap(400)
```

```
False
True
False
True
```

```
# 另外一个更为简洁的版本，理解它还挺练脑子的
# cpython/Lib/datetime.py
def _is_leap(year):
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
_is_leap(300)
```

```
False
```

函数可以同时接收多个参数。比如，我们可以写个函数，让它输出从大于某个数字到小于另外一个数字的斐波那契数列；那就需要定义两个参数，调用它的时候也需要传递两个参数：

```
def fib_between(start, end):
    a, b = 0, 1
    while a < end:
        if a >= start:
            print(a, end=' ')
        a, b = b, a + b
```

```
fib_between(100, 10000)
```

```
144 233 377 610 987 1597 2584 4181 6765
```

当然可以把这个函数写成返回值是一个列表：

```
def fib_between(start, end):
    r = []
    a, b = 0, 1
    while a < end:
        if a >= start:
            r.append(a)
        a, b = b, a + b
    return r

fib_between(100, 10000)
```

```
[144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

变量的作用域

下面的代码，经常会让初学者迷惑：

```
def increase_one(n):
    n += 1
    return n

n = 1
print(increase_one(n))
# print(n)
```

```
2
```

当 `increase_one(n)` 被调用之后，`n` 的值究竟是多少呢？或者更准确点问，随后的 `print(n)` 的输出结果应该是什么呢？

输出结果是 `1`。

在程序执行过程中，变量有**全局变量**（Global Variables）和**局域变量**（Local Variables）之分。

> 首先，每次某个函数被调用的时候，这个函数会开辟一个新的区域，这个函数内部所有的变量，都是局域变量。也就是说，即便那个函数内部某个变量的名称与它外部的某个全局变量名称相同，它们也不是同一个变量 —— 只是名称相同而已。
>
> 其次，更为重要的是，当外部调用一个函数的时候，准确地讲，传递的不是变量，而是那个变量的*值*。也就是说，当 `increase_one(n)` 被调用的时候，被传递给那个恰好名称也叫 `n` 的局域变量的，是全局变量 `n` 的值，`1`。
>
> 而后，`increase_one()` 函数的代码开始执行，局域变量 `n` 经过 `n += 1` 之后，其中存储的值是 `2`，而后这个值被 `return` 语句返回，所以，`print(increase(n))` 所输出的值是函数被调用之后的返回值，即，`2`。
>
> 然而，全局变量 `n` 的值并没有被改变，因为局部变量 `n`（它的值是 `2`） 和全局变量 `n`（它的值还是 `1`）只不过是名字相同而已，但它们并不是同一个变量。

以上的文字，可能需要反复阅读若干遍；几遍下来，消除了疑惑，以后就彻底没问题了；若是这个疑惑并未消除，或者关键点并未消化，以后则会反复被这个疑惑所坑害，浪费无数时间。

不过，有一种情况要格外注意 —— 在函数内部处理被传递进来的值是可变容器（比如，列表）的时候：

```
def be_careful(a, b):
    a = 2
    b[0] = 'What?!'

a = 1
b = [1, 2, 3]
be_careful(a, b)
a, b
```

```
(1, ['What?!', 2, 3])
```

所以，一个比较好的习惯是，如果传递进来的值是列表，那么在函数内部对其操作之前，先创建一个它的拷贝：

```
def be_careful(a, b):
    a = 2
    b_copy = b.copy()
    b_copy[0] = 'What?!'

a = 1
b = [1, 2, 3]
be_careful(a, b)
a, b
```

```
(1, [1, 2, 3])
```