第 3 章

控制语句和函数

Python 除了拥有进行基本运算的能力，同时也具有写出一个完整程序的能力，那么对于程序中各种复杂的逻辑该怎么控制呢？这就到了控制语句派上用场的时候了。

对于一个结构化的程序来说，一共只有三种执行结构，如果用圆角矩形表示程序的开始和结束，直角矩形表示执行过程，菱形表示条件判断，那么三种执行结构可以分别用如图 3-1～图 3-3 所示的三张图表示。

顺序结构：就是做完一件事后紧接着做另一件事，如图 3-1 所示。

选择结构：在某种条件成立的情况下做某件事，反之做另一件事，如图 3-2 所示。

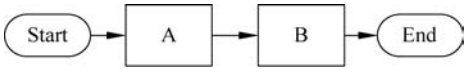

图 3-1　顺序结构

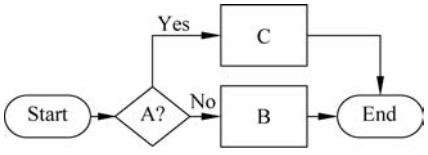

图 3-2　选择结构

循环结构：反复做某件事，直到满足某个条件为止，如图 3-3 所示。

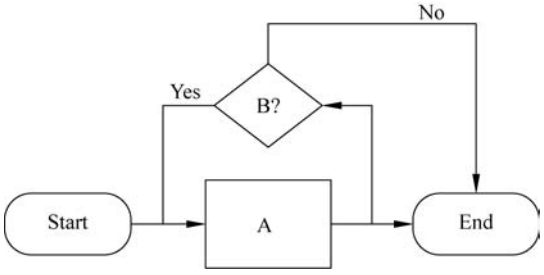

图 3-3　循环结构

程序语句的执行默认就是顺序结构，而条件结构和循环结构分别对应条件语句和循环语句，它们都是控制语句的一部分。

那什么是控制语句呢？这个词出自 C 语言，对应的英文是 Control Statements。它的作用是控制程序的流程，以实现各种复杂逻辑。下面将重点介绍 Python 中实现顺序结构、选择结构、循环结构的语法。

3.1 选择结构

在 Python 中，选择结构的实现是通过 if 语句，if 语句的常见语法是：

```
if 条件 1:
    代码块 1
elif 条件 2:
    代码块 2
elif 条件 3:
    代码块 3
    …
    …
elif 条件 n - 1:
    代码块 n - 1
else
    代码块 n
```

这表示的是，如果条件 1 成立就执行代码块 1，接着如果条件 1 不成立而条件 2 成立就执行代码块 2，如果条件 1 到条件 n－1 都不满足，那么就执行代码块 n。

另外其中的 elif 和 else 以及相应的代码块是可以省略的，也就是说最简单的 if 语句格式是：

```
if 条件:
  代码段
```

要注意的是，这里所有代码块前应该是 4 个空格，原因稍后会提到，这里先看一段具体的 if 语句。

```
a = 4
if a < 5:
   print('a is smaller than 5.')
elif a < 6:
   print('a is smaller than 6.')
else:
   print('a is larger than 5.')
```

很容易得到结果：

```
a is smaller than 5.
```

这段代码表示的含义就是，如果 a 小于 5 则输出'a is smaller than 5.'，如果 a 不小于 5 而小于 6 则输出'a is smaller than 6.'，否则就输出'a is larger than 5.'。这里值得注意的一点是，虽然 a 同时满足 a＜5 和 a＜6 两个条件，但是由于 a＜5 在前面，所以最终输出的为'a is smaller than 5.'。

if 语句的语义非常直观易懂，但是这里还有一个问题没有解决，那就是为什么要在代码块之前空 4 格？

依旧是先看一个例子：

```
if 1 > 2:
    print('Impossible!')
print('done')
```

运行这段代码可以得到：

```
done
```

但是如果稍加改动，在 print('done')前也加 4 个空格：

```
if 1 > 2:
     print('Impossible!')
     print('done')
```

再运行的话什么也不会输出。

它们的区别是什么呢？对于第一段代码，print('done')和 if 语句是在同一个代码块中的，也就是说无论 if 语句的结果如何 print('done')一定会被执行。而在第二段代码中 print('done') 和 print('Impossible!') 在同一个代码块中的，也就是说如果 if 语句中的条件不成立，那么，print('Impossible!')和 print('done')都不会被执行。

我们称第二个例子中这种拥有相同的缩进的代码为一个代码块。虽然 Python 解释器支持使用任意多但是数量相同的空格或者制表符来对齐代码块，但是一般约定用 4 个空格作为对齐的基本单位。

另外值得注意的是，在代码块中是可以再嵌套另一个代码块的，以 if 语句的嵌套为例：

```
a = 1
b = 2
c = 3
if a > b:   #第 4 行
    if a > c:
        print('a is maximum.')
    elif c > a:
        print('c is maximum.')
    else:
```

```
        print('a and c are maximum.')
    elif a < b:   #第 11 行
        if b > c:
            print('b is maximum.')
        elif c > b:
            print('c is maximum.')
        else:
            print('b and c are maximum.')
    else:   #第 19 行
        if a > c:
            print('a and b are maximum')
        elif a < c:
            print('c is maximum')
        else:
            print('a, b, and c are equal')
```

首先最外层的代码块是所有的代码，它的缩进是 0，接着它根据 if 语句分成了 3 个代码块，分别是第 5～10 行，第 12～18 行，第 20～27 行，它们的缩进是 4，接着在这 3 个代码块内又根据 if 语句分成了 3 个代码块，其中每个 print 语句是一个代码块，它们的缩进是 8。

从这个例子中我们可以看到代码块是有层级的、是嵌套的，所以即使这个例子中所有的 print 语句拥有相同的空格缩进，仍然不是同一个代码块。

但是单有顺序结构和选择结构是不够的，有时候某些逻辑执行的次数本身就是不确定的或者说逻辑本身具有重复性，那么这时候就需要循环结构了。

3.2 循环结构

Python 的循环结构有两个关键字可以实现，分别是 while 和 for。

3.2.1 while 循环

while 循环的常见语法是：

```
while 条件:
代码块
```

这个代码块表达的含义就是，如果条件满足就执行代码块，直到条件不满足为止，如果条件一开始不满足那么代码块一次都不会被执行。

我们看一个例子：

```
a = 0
while a < 5:
    print(a)
    a += 1
```

运行这段代码可以得到输出如下：

```
0
1
2
3
4
```

对于 while 循环，其实和 if 语句的执行结构非常接近，区别就是从单次执行变成了反复执行，以及条件除了用来判断是否进入代码块以外还被用来作为是否终止循环的判断。

对于上面这段代码，结合输出我们不难看出，前五次循环的时候 a＜5 为真因此循环继续，而第六次经过的时候，a 已经变成了 5，条件就为假，自然也就跳出了 while 循环。

3.2.2　for 循环

for 循环的常见语法是：

```
for 循环变量 in 可迭代对象:
代码段
```

Python 的 for 循环比较特殊，它并不是 C 系语言中常见的 for 语句，而是一种 foreach 的语法，也就是说本质上是遍历一个可迭代的对象，这听起来实在是太抽象了，我们看一个例子：

```
for i in range(5):
    print(i)
```

运行后这段代码输出如下：

```
0
1
2
3
4
```

for 循环实际上用到了迭代器的知识，但是在这里展开还为时尚早，我们只要知道用 range 配合 for 可以写出一个循环即可，例如计算 0～100 整数的和：

```
sum = 0
for i in range(101):   #别忘了 range(n)的范围是[0, n-1)
    sum += i
print(sum)
```

那如果想计算 50～100 整数的和呢？实际上 range 产生区间的左边界也是可以设置

的，只要多传入一个参数：

```
sum = 0
for i in range(50, 101):   #range(50 ,101) 产生的循环区间是 [50, 101)
    sum += i
print(sum)
```

有时候我们希望循环是倒序的，例如从 10 循环到 1，那该怎么写呢？只要再多传入一个参数作为步长即可：

```
for i in range(10, 0, -1): #这里循环区间是 (1, 10],但是步长是 -1
    print(i)
```

也就是说 range 的完整用法应该是 range(start,end,step)，循环变量 i 从 start 开始，每次循环后 i 增加 step 直到超过 end 跳出循环。

3.2.3 两种循环的转换

其实无论是 while 循环还是 for 循环，本质上都是反复执行一段代码，这就意味着二者是可以相互转换的，例如之前计算整数 0～100 的代码，也可以用 while 循环完成，如下所示：

```
sum = 0
i = 0
while i <= 100:
    sum += i
    i ++
print(sum)
```

但是这样写之后至少存在三个问题：

① while 写法中的条件为 i <=100，而 for 写法是通过 range()来迭代，相比来说后者显然更具可读性。

② while 写法中需要在外面创建一个临时的变量 i，这个变量在循环结束依旧可以访问，但是 for 写法中 i 只有在循环体中可见，明显 while 写法增添了不必要的变量。

③ 代码量增加了两行。

当然这个问题是辩证性的，有时候 while 写法可能是更优解，但是对于 Python 来说，大多时候推荐使用 for 这种可读性强也更优美的代码。

3.3 break、continue 与 pass

学习了三种基本结构，我们已经可以写出一些有趣的程序了，但是 Python 还有一些控制语句可以让代码更加优美简洁。

3.3.1　break 与 continue

break 和 continue 只能用在循环体中，通过一个例子来了解一下其作用：

```
i = 0
while i <= 50:
    i += 1
    if i == 2:
        continue
    elif i == 4:
        break
    print(i)
print('done')
```

这段代码会输出：

```
1
3
done
```

这段循环中如果没有 continue 和 break 的话应该是输出 1～51 的，但是这里输出只有 1 和 3，为什么呢？

我们首先考虑当 i 为 2 的那次循环，它进入了 if i==2 的代码块中，执行了 continue，这次循环就被直接跳过了，也就是说后面的代码包括 print(i)都不会再被执行，而是直接进入了下一次 i=3 的循环。

接着考虑当 i 为 4 的那次循环，它进入了 elif i == 4 的代码块中，执行了 break，直接跳出了循环到最外层，然后接着执行循环后面的代码输出了 done。

所以总结一下，continue 的作用是跳过剩下的代码进入下一次循环，break 的作用是跳出当前循环然后执行循环后面的代码。

这里有一点需要强调的是，break 和 continue 只能对当前循环起作用，也就是说如果在循环嵌套的情况下想对外层循环起控制作用，需要多个 break 或者 continue 联合使用。

3.3.2　pass

pass 很有意思，它的功能就是没有功能。看一个例子：

```
a = 0
if a >= 10:
    pass
else:
    print('a is smaller than 10')
```

我们要想在 a>10 的时候什么都不执行，但是如果什么不写的话又不符合 Python 的缩进要求，为了使得语法上正确，我们这里使用了 pass 来作为一个代码块，但是 pass 本身不

会有任何效果。

3.4 函数的定义与使用

还记得我们上一章提到过的一个“内置函数”max 吗？对于不同的 List 和 Tuple，这个函数总能给出正确的结果。当然有人说，用 for 循环实现也很快很方便，但是有多少个 List 或 Tuple 就要写多少个完全重复的 for 循环，这是很让人厌烦的，这时候就需要函数出场了。

本章会从数学中的函数引入，详细讲解 Python 中函数的基本用法。

3.4.1 认识 Python 的函数

函数的数学定义为：给定一个数集 A，对 A 施加对应法则 f，记作 $f(A)$，得到另一数集 B，也就是 $B=f(A)$，那么这个关系式就叫函数关系式，简称函数。

数学中的函数其实就是 A 和 B 之间的一种关系，可以理解为从 A 中取出任意一个输入都能在 B 中找到特定的输出，在程序中，函数也是完成这样的一种输入到输出的映射，但是程序中的函数有着更大的意义。

它首先可以减少重复代码，因为可以把相似的逻辑抽象成一个函数，减少重复代码，其次它有可以使程序模块化并且提高可读性。

以之前多次用到的一个函数 print 为例：

```
print('Hello, Python!')
```

由于 print 是一个函数，因此不用再去实现一遍打印到屏幕的功能，减少了大量的重复代码，同时看到 print 就可以知道这一行是用来打印的，可读性自然就提高了。另外如果打印出现问题，只要去查看 print 函数的内部就可以了，而不用再去看 print 以外的代码，这体现了模块化的思想。

但是，内置函数的功能非常有限，需要根据实际需求编写我们自己的函数，这样才能进一步提高程序的简洁性、可读性和可扩展性。

3.4.2 函数的定义和调用

1. 定义

和数学中的函数类似，Python 中的函数需要先定义才能使用，例如：

```
def ask_me_to(string):
print(f'You want me to {string}?')
if string == 'swim':
    return 'OK!'
else:
    return "Don't even think about it."
```

这是一个基本的函数定义，其中第 1、4、6 行是函数特有的，其他我们都已经学习过了。

我们先看第 1 行：

```
def ask_me_to(string):
```

这一行有四个关键点：

① def：函数定义的关键字，写在最前面。

② ask_me_to：函数名，命名要求和变量一致。

③ (string)：函数的参数，多个参数用逗号隔开。

④ 结尾冒号：函数声明的语法要求。

然后第 2～5 行：

```
print(f'You want me to {string}?')
if string == 'swim':
    return 'OK!'
else:
    return "Don't even think about it."
```

它们都缩进了四个空格，意味着它们构成了一个代码块，同时从第 2 行可以看到函数内是可以接着调用函数的。

我们接着再看第 4 行。

```
return 'OK!'
```

这里引入了一个新关键字：return，它的作用是结束函数并返回到之前调用函数处的下一句。返回的对象是 return 后面的表达式，如果表达式为空则返回 None。第 6 行跟第 4 行功能相同，这里不再赘述。

2. 调用

在数学中函数需要一个自变量才会得到因变量，Python 的函数也是一样，只是定义的话并不会执行，还需要调用，例如：

```
print(ask_me_to('dive'))
```

注意这里是两个函数嵌套，首先调用的是自定义的函数 ask_me_to，接着 ask_me_to 的返回值传给了 print，所以会输出 ask_me_to 的返回值：

```
You want me to dive?
Don't even think about it.
```

定义和调用都很好理解，接下来看看函数的参数怎么设置。

3.4.3　函数的参数

Python 的函数参数非常灵活，我们已经学习了最基本的一种，例如：

```
def ask_me_to(string):
```

它拥有一个参数，名字为 string。

函数参数的个数可以为 0 个或多个，例如：

```
def random_number():
    return 4 #刚用骰子扔的，绝对随机
```

可以根据需求去选择参数个数，但是要注意的是即使没有参数，括号也不可省略。

Python 的一个灵活之处在于函数参数形式的多样性，有以下几种：

- 不带默认参数的：def func(a)。
- 带默认参数的：def func(a, b=1)。
- 任意位置参数：def func(a, b=1, *c)。
- 任意键值参数：def func(a, b=1, *c, **d)。

第一种就是刚才讲到的一般形式，来看一看剩下三种如何使用。

3.4.4 默认参数

有时候某个函数参数大部分时候为某个特定值，于是我们希望这个参数可以有一个默认值，这样就不用频繁指定相同的值给这个参数了。默认参数的用法看一个例子：

```
def print_date(year, month=1, day=1):
    print(f'{year:04d}-{month:02d}-{day:02d}')
```

这是一个格式化输出日期的函数，注意其中月份和天数参数我们用一个等号表明赋予默认值。于是，可以分别以 1,2,3 个参数调用这个函数，同时也可以指定某个特定参数，例如：

```
print_date(2018)
print_date(2018, 2, 1)
print_date(2018, 5)
print_date(2018, day=3)
print_date(2018, month=2, day=5)
```

这段代码会输出：

```
2018-01-01
2018-02-01
2018-05-01
2018-01-03
2018-02-05
```

我们依次看一下这些调用。

① print_date(2018)这种情况下由于默认参数的存在等价于 print_date(2018, 1, 1)。

② print_date(2018，2，1)这种情况下所有参数都被传入了，因此和无默认参数的行为是一致的。

③ print_date(2018，5)省略了 day，因为参数是按照顺序传入的。

④ print_date(2018，day＝3)省略了 month，由于和声明顺序不一致，所以必须声明参数名称。

⑤ print_date(2018，month＝2，day＝5)全部声明也是可以的。

使用默认参数可以让函数的行为更加灵活。

3.4.5　任意位置参数

如果函数想接收任意数量的参数，那么可以这样声明使用：

```
def print_args( * args):
    for arg in args:
        print(arg)

print_args(1, 2, 3, 4)
```

诊断代码会输出：

```
1
2
3
4
```

任意位置参数的特点就是它只占一个参数，并且以 * 开头。其中 args 为一个 List，包含了所有传入的参数，顺序为调用时候传参的顺序。

3.4.6　任意键值参数

除了接收任意数量的参数，如果希望给每个参数一个名字，那么可以这么声明参数：

```
def print_kwargs( ** kwargs):
    for kw in kwargs:
        print(f'{kw} = {kwargs[kw]}')

print_kwargs(a = 1, b = 2, c = 3, d = 4)
```

这段代码会输出：

```
a = 1
b = 2
c = 3
d = 4
```

跟之前讲过的任意位置参数使用非常类似，但是 kwargs 这里是一个 Dict，其中 Key 和 Value 为调用时候传入的参数名称和值，顺序和传参顺序一致。

3.4.7 组合使用

现在知道了这四类参数可以同时使用，但是需要满足一定的条件，例如：

```
def the_ultimate_print_args(arg1, arg2 = 1, *args, **kwargs):
    print(arg1)
    print(arg2)
    for arg in args:
        print(arg)
    for kw in kwargs:
        print(f'{kw} = {kwargs[kw]}')
```

可以看出，四种参数在定义时应该满足这样的顺序：非默认参数、默认参数、任意位置参数、任意键值参数。

调用的时候，参数分两类，即位置相关参数和无关键词参数，例如：

```
the_ultimate_print_args(1, 2, 3, arg4 = 4)  #1,2,3 是位置相关参数,arg4 = 4 是关键词参数
```

这句代码会输出：

```
1
2
3
arg4 = 4
```

其中前三个就是位置相关参数，最后一个是关键词参数。位置相关参数是顺序传入的，而关键词参数则可以乱序传入，例如：

```
the_ultimate_print_args(arg3 = 3, arg2 = 2, arg1 = 3, arg4 = 4)  #这里 arg1 和 arg2 是乱序的!
```

这句代码会输出：

```
3
2
arg3 = 3
arg4 = 4
```

总之在调用的时候参数顺序应该满足的规则是：

① 位置相关参数不能在关键词参数之后。

② 位置相关参数优先。

这么看太抽象，不如看看两个错误用法。第一个错误用法：

```
the_ultimate_print_args(arg4 = 4, 1, 2, 3)
```

这句代码会报错：

```
Traceback (most recent call last):
   File  "/Users/jiangjiao/PycharmProjects/LearnPythonWithPractice/Chapter  8/Parameters.
py", line 43
    the_ultimate_print_args(arg4 = 4, 1, 2, 3)
                                      ^
SyntaxError: positional argument follows keyword argument
```

报错的意思是位置相关参数不能在关键词参数之后。也就是说，必须先传入位置相关参数，再传入关键词参数。

再看第二个错误用法：

```
the_ultimate_print_args(1, 2, arg1 = 3, arg4 = 5)
```

这句代码会报错：

```
Traceback (most recent call last):
   File  "/Users/jiangjiao/PycharmProjects/LearnPythonWithPractice/Chapter  8/Parameters.
py", line 41, in <module>
    the_ultimate_print_args(1, 2, arg1 = 3, arg4 = 5)
TypeError: the_ultimate_print_args() got multiple values for argument 'arg1'
```

报错意思是函数的参数 arg1 接收到了多个值。也就是说，位置相关参数会优先传入，如果再指定相应的参数那么就会发生错误。

3.4.8　修改传入的参数

先补充有关传入参数的两个重要概念。

① 按值传递：复制传入的变量，传入函数的参数是一个和原对象无关的副本。

② 按引用传递：直接传入原变量的一个引用，修改参数就是修改原对象。

在有些编程语言中，可能是两种传参方式同时存在可供选择，但是 Python 只有一种传参方式就是按引用传递，例如：

```
list1 = [1, 2, 3]
def new_element(mylist):
   mylist.append(4)   #mylist 是一个引用!

new_element(list1)
print(list1)
```

注意我们在函数内通过 append()修改了 mylist 的元素，由于 mylist 是 list1 的一个引

用,因此实际上我们修改的就是 list1 的元素,所以这段代码会输出:

```
[1, 2, 3, 4]
```

这是符合我们的预期的,但是我们看另一个例子:

```
num = 1
def edit_num(number):
    number += 2
edit_num(num)
print(num)
```

按照之前的理论,number 应该是 num 的一个引用,所以这里应该输出 3,但是实际上输出是:

```
1
```

为什么会这样呢? 在第 6 章会讲到:特别地,字符串是一个不可变的对象。实际上,包括字符串在内,数值类型和 Tuple 也是不可变的,而这里正是因为 num 是不可变类型,所以函数的行为不符合我们的预期。

为了深入探究其原因,引入一个新的内建函数 id,它的作用是返回对象的 id。对象的 id 是唯一的,但是可能有多个变量引用同一个对象,例如下面这个例子:

```
alice = 32768
bob = alice  #看起来我们赋值了
print(id(alice))
print(id(bob))
alice += 1   #这里要修改 alice
print(id(alice))
print(id(bob))
print(alice)
print(bob)
```

可以得到这样的输出(这里 id 的输出不一定跟本书一致,但是第 1,2,4 个 id 应该是相同的):

```
4320719728
4320719728
4320720144
4320719728
32769
32768
```

其实除了函数参数是引用传递,Python 变量的本质就是引用。这也就意味着我们在把 alice 赋值给 bob 的时候,实际上是把 alice 的引用给了 bob,于是这时候 alice 和 bob 实际上

引用了同一个对象，因此 id 相同。

接下来，我们修改了 alice 的值，可以看到 bob 的值并没有改变，这符合我们的直觉。但是从引用上看，实际发生的操作是，bob 的引用不变，但是 alice 获得了一个新对象的引用，这个过程充分体现了数值类型不可变的性质——已经创建的对象不会修改，任何修改都是新建一个对象来实现。

实际上，对于这些不可变类型，每次修改都会创建一个新的对象，然后修改引用为新的对象。在这里，alice 和 bob 已经引用两个完全不同的对象了，这两个对象占用的空间是完全不同的。

那么回到最开始的问题，为什么这些不可变对象在函数内的修改不能体现在函数外呢？虽然函数参数的确引用了原对象，但是我们在修改的时候实际上是创建了一个新的对象，所以原对象不会被修改，这也就解释了刚才的现象。如果一定要修改的话，可以这么写：

```
num = 1
def edit_num(number):
    number += 2
    return number
num = edit_num(num)
print(num)   # 会输出 3
```

这样输出就是我们预期的 3 了。

特殊地，这里举例用了一个很大的数字是有原因的。由于 0～256 这些整数使用得比较频繁，为了避免小对象的反复内存分配和释放造成内存碎片，所以 Python 对 0～256 这些数字建立了一个对象池。

```
alice = 1
bob = 1
print(id(alice))
print(id(bob))
```

可以得到输出为（这里输出的两个 id 应该是一致的，但是数字不一定跟本书中的相同）：

```
4482894448
4482894448
```

可以看出，虽然 alice 和 bob 无关，但是它们引用的是同一个对象，所以为了方便说明，之前取了一个比较大的数字用于赋值。

3.4.9　函数的返回值

1. 返回一个值

函数在执行的时候，会在执行到结束或者 return 语句的时候返回调用的位置。如果我

们的函数需要返回一个值，那需要用 return 语句，例如最简单的，返回一个值：

```
def multiply(num1, num2):
    return num1 * num2

print(multiply(3, 5))
```

这段代码会输出：

```
15
```

这个 multiply 函数将输入的两个参数相乘，然后返回结果。

2. 什么都不返回

如果我们不想返回任何内容，可以只写一个 return，它会停止执行后面代码的立即返回，例如：

```
def guess_word(word):
    if word != 'secret':
        return  # 等价于 return None
    print('bingo')

guess_word('absolutely not this one')
```

这里只要函数参数不是'secret'就不会输出任何内容，因为 return 后面的代码不会被执行。另外 return 跟 return None 是等价的，也就是说默认返回的是 None。

3. 返回多个值

和大部分编程语言不同，Python 支持返回多个参数，例如：

```
def reverse_input(num1, num2, num3):
    return num3, num2, num1

a, b, c = reverse_input(1, 2, 3)
print(a)
print(b)
print(c)
```

这里要注意接收返回值的时候不能再像之前用一个变量，而是要用和返回值数目相同的变量接收，其中返回值赋值的顺序是从左到右的，跟直觉一致。

```
3
2
1
```

所以这个函数的作用就是把输入的三个变量顺序翻转一下。

3.4.10　函数的嵌套

可以在函数内定义函数，这对于简化函数内重复逻辑很有用，例如：

```
def outer():
    def inner():
        print('this is inner function')
    print('this is outer function')
    inner()

outer()
```

这段代码会输出：

```
this is outer function
this is inner function
```

需要注意的一点是，内部的函数只能在它所处的代码块中使用，在上面这个例子中，inner 在 outer 外面是不可见的，这个概念称为作用域。

1. 作用域

作用域是一个很重要的概念，看一个例子：

```
def func1():
    x1 = 1

def func2():
    print(x1)

func1()
func2()
```

这里函数 func2 中能正常输出 x1 的值吗？

答案是不能。为了解决这个问题，需要学习 Python 的变量名称查找顺序，即 LEGB 原则。

① L：Local(本地)是函数内的名字空间，包括局部变量和形参。

② E：Enclosing(封闭)外部嵌套函数的名字空间(闭包中常见)。

③ G：Global(全局)全局函数定义所在模块的名字空间。

④ B：Builtin(内建)内置模块的名字空间。

LEGB 原则的含义是，Python 会按照 LEGB 这个顺序去查找变量，一旦找到就拿来用，否则就到更外面一层的作用域去查找，如果都找不到就报错。

可以通过一个例子来认识 LEGB，例如：

```
a = 1              #对于 func3 和 inner 来说都是 Global
def func3():
    b = 2          #对于 func3 来说是 Local,对于 inner 来说是 Enclosing
    def inner():
        c = 3      #对于 inner 来说是 Local,func3 不可见
```

其中要注意的是 func3 没有 Enclosing 作用域,至于闭包是什么我们会在后面的章节中见到,这里只要理解 LEGB 原则就可以了。

2. global 和 nonlocal

根据上述 LEGB 原则,在函数中是可以访问到全局变量的,例如:

```
d = 1
def func4():
    d += 2

func4()
```

但是 LEGB 规则仿佛出了点问题,因为会报错:

```
Traceback (most recent call last):
  File "/Users/jiangjiao/PycharmProjects/LearnPythonWithPractice/Chapter 8/Function within
Function.py", line 36, in <module>
    func4()
  File "/Users/jiangjiao/PycharmProjects/LearnPythonWithPractice/Chapter 8/Function within
Function.py", line 33, in func4
    d += 2
UnboundLocalError: local variable 'd' referenced before assignment
```

这并不是 Python 的问题,反而是 Python 的一个特点,也就是说 Python 阻止用户在不自觉的情况下修改非局部变量,那么怎么访问非局部变量呢?

为了修改非局部变量,我们需要使用 global 和 nonlocal 关键字,其中,nonlocal 关键词是 Python 3 中才有的新关键词,看一个例子:

```
d = 1
def func4():
    global d
    e = 5
    d += 2             #访问到了全局变量 d
    def inner():
        nonlocal e
        e += 3         #访问到了闭包中的变量 e
    inner()
    print(e)

func4()
print(d)
```

也就是说，global 会使得相应的全局变量在当前作用域内可见，而 nonlocal 可以让闭包中非全局变量可见，所以这段代码会输出：

```
8
3
```

3.4.11　使用轮子

这里的“使用轮子”可不是现实中那种使用轮子，而是指直接使用别人写好封装好的易于使用的库，进而极大地减少重复劳动，提高开发效率。

Python 自带的标准库就是一堆鲁棒性强、接口易用、涉猎广泛的“轮子”，善于利用这些轮子可以极大地简化代码，这里简单介绍一些常用的库。

1. 随机库

Python 中的随机库用于生成随机数，例如：

```
import random  # 之前 return 4 那个只是开个玩笑
print(random.randint(1, 5))
```

它会输出一个随机的 [1,5) 范围内的整数。无须关心它的实现，只要知道这样可以生成随机数就可以了。

其中，import 关键字的作用是导入一个包，有关包和模块的内容后面章节会细讲，这里只讲基本使用方法。

用 import 导入的基本语法是：import 包名，包提供的函数的用法是 包名.函数名。当然不仅函数，包里面的常量和类都可以通过类似的方法调用，不过这里会用函数就够了。

此外如果不想写包名，也可以这样：

```
from random import randint
```

然后就可以直接调用 randint 而不用写前面的 random. 了。

如果有很多函数要导入的话，还可以这么写：

```
from random import *
```

这样 random 包里的一切就都包含进来了，可以不用 random. 直接调用。不过不太推荐这样写，因为不知道包内都有什么，容易造成名字的混乱。

特殊地，import random 还有一种特殊写法：

```
import random as rnd
print(rnd.randint(1, 5))
```

它和 import random 没有本质区别，仅仅是给了 random 一个方便输入的别名 rnd。

2. 日期库

这个库可以用于计算日期和时间，例如：

```
import datetime
print(datetime.datetime.now())
```

这段代码会输出：

```
2018-04-29 20:40:21.164027
```

3. 数学库

这个库有着常用的数学函数，例如：

```
import math
print(math.sin(math.pi / 2))
print(math.cos(math.pi / 2))
```

这段代码会输出：

```
1.0
6.123233995736766e-17
```

其中第二个结果其实就是 0，但是限于浮点数的精度问题无法精确表示为 0，所以在编写代码涉及浮点数比较的时候一定要这么写：

```
EPS = 1e-8
print(abs(math.cos(math.pi / 2)) < EPS)
```

这里 EPS 就是指允许的误差范围。也就是说浮点数没有真正的相等，只有在一定误差范围内的相等。

4. 操作系统库

这个库包含操作系统的一些操作，例如列出目录：

```
import os
print(os.listdir('.'))
```

我们在之后的文件操作章节还会见到这个库。

5. 第三方库

还记得我们第 3 章讲过的 pip 吗，可以用 pip 来方便的安装各种第三方库，例如：

```
pip install numpy
```

通过一行指令就可以安装 numpy 这个库了，然后就可以在代码中正常 import 这个库：

```
import numpy
```

这也正是 pip 作为包管理器强大的地方，方便易用。

本章小结

本章介绍了三种执行结构和 Python 的控制语句，并且引入了代码块这个重要的概念，只要完全掌握这些内容，理论上就可以写出任何程序了，所以一定要在理解的基础上熟练使用 Python 的各种控制语句，打下良好的基础。

通过本章的学习我们还看到 Python 的函数定义简单，而且无论是在参数设置上还是结果返回上都具有极高的灵活性，同时借助函数也接触到了“作用域”这个重要的概念，最后学习了库的简单使用。善用函数，往往可以使代码更加简洁优美。

本章习题

1. 通过选择结构把一门课的成绩转化成绩点并输出，其中成绩点的计算为了简单起见，采用 90～100 分 4.0，80～89 分 3.0，70～79 分 2.0，60～69 分 1.0 的规则。

2. 给定一个分段函数，在 $x \geqslant 0$ 的时候，$y=x$；在 $x<0$ 的时候；为 $x=0$，实现这个函数的计算逻辑。

3. 给定三个整数 a，b，c，判断哪个最小。

4. 使用循环计算 1～100 中所有偶数的和。

5. 水仙花数是指一个 n 位数（$n \geqslant 3$），它的每个位上的数字的 n 次幂之和等于它本身。输出所有三位数水仙花数。

6. 斐波那契数列是一个递归定义的数列，它的前两项为 1，从第三项开始每项都是前面两项的和。输出 100 以内的斐波那契数列。

7. 输入一个数字，判断它在不在斐波那契数列中。

8. 通过自学递归的概念，构造一个递归函数实现斐波那契数列的计算。

9. 通过使用默认参数，实现可以构造一个等差数列的函数，参数包括等差数列的起始、结束以及公差，注意公差应该可以为负数。

10. 写一个日期格式化函数，使用键值对传递参数。

11. 实现能够返回 List 中第 n 大的数字的函数，n 由输入指定。

12. 写一个函数，求两个数的最大公约数。

13. 通过循环和函数，写一个井字棋游戏，并写一个井字棋的 AI。

14. 查询日期库文档，写代码完成当前时间从 UTC+8（北京时间）到 UTC−5 的转换。

15. 查询随机库文档，写一个投骰子程序，要求可以指定骰子面数和数量，并计算投掷的数学期望。