**1. for - else**

什么？不是 if 和 else 才是原配吗？No，你可能不知道，else 是个脚踩两只船的家伙，for 和 else 也是一对，而且是合法的。

>>> for i in [1,2,3,4]:

print(i)

else:

print(i, '我是else')

1

2

3

4

4 我是else

如果在 for 和 else 之间（循环体内）有第三者 if 插足，也不会影响 for 和 else 的关系。因为 for 的级别比 if 高，else 又是一个攀附权贵的家伙，根本不在乎是否有 if，以及是否执行了满足 if 条件的语句。else 的眼里只有 for，只要 for 顺利执行完毕，else 就会屁颠儿屁颠儿地跑一遍：

>>> for i in [1,2,3,4]:

if i > 2:

print(i)

else:

print(i, '我是else')

3

4

4 我是else

那么，如何拆散 for 和 else 这对冤家呢？只有当 for 循环被 break 语句中断之后，才会跳过 else 语句：

>>> for i in [1,2,3,4]:

if i>2:

print(i)

break

else:

print(i, '我是else')

3

**2. 一颗星(\*)和两颗星(\*\*)**

有没有发现，星(\*)真是一个神奇的符号！想一想，没有它，C语言还有啥好玩的？同样，因为有它，Python 才会如此的仪态万方、风姿绰约、楚楚动人！Python 函数支持默认参数和可变参数，一颗星表示不限数量的单值参数，两颗星表示不限数量的键值对参数。

我们还是举例说明吧：设计一个函数，返回多个输入数值的和。我们固然可以把这些输入数值做成一个list传给函数，但这个方法，远没有使用一颗星的可变参数来得优雅：

>>> def multi\_sum(\*args):

s = 0

for item in args:

s += item

return s

>>> multi\_sum(3,4,5)

12

Python 函数允许同时全部或部分使用固定参数、默认参数、单值（一颗星）可变参数、键值对（两颗星）可变参数，使用时必须按照前述顺序书写。

>>> def do\_something(name, age, gender='男', \*args, \*\*kwds):

print('姓名：%s，年龄：%d，性别：%s'%(name, age, gender))

print(args)

print(kwds)

>>> do\_something('xufive', 50, '男', 175, 75, math=99, english=90)

姓名：xufive，年龄：50，性别：男

(175, 75)

{'math': 99, 'english': 90}

此外，一颗星和两颗星还可用于列表、元组、字典的解包，看起来更像C语言：

>>> a = (1,2,3)

>>> print(a)

(1, 2, 3)

>>> print(\*a)

1 2 3

>>> b = [1,2,3]

>>> print(b)

[1, 2, 3]

>>> print(\*b)

1 2 3

>>> c = {'name':'xufive', 'age':51}

>>> print(c)

{'name': 'xufive', 'age': 51}

>>> print(\*c)

name age

>>> print('name:{name}, age:{age}'.format(\*\*c))

name:xufive, age:51

**3. 三元表达式**

熟悉 C/C++ 的程序员，初上手 python 时，一定会怀念经典的三元操作符，因为想表达同样的思想，用python 写起来似乎更麻烦。比如：

>>> y = 5

>>> if y < 0:

print('y是一个负数')

else:

print('y是一个非负数')

y是一个非负数

其实，python 是支持三元表达式的，只是稍微怪异了一点，类似于我们山东人讲话。比如，山东人最喜欢用倒装句：打球去吧，要是不下雨的话；下雨，咱就去自习室。翻译成三元表达式就是：

打球去吧 if 不下雨 else 去自习室

来看看三元表达式具体的使用：

>>> y = 5

>>> print('y是一个负数' if y < 0 else 'y是一个非负数')

y是一个非负数

python 的三元表达式也可以用来赋值：

>>> y = 5

>>> x = -1 if y < 0 else 1

>>> x

1

**4. with - as**

with 这个词儿，英文里面不难翻译，但在 Python 语法中怎么翻译，我还真想不出来，大致上是一种上下文管理协议。作为初学者，不用关注 with 的各种方法以及机制如何，只需要了解它的应用场景就可以了。with 语句适合一些事先需要准备，事后需要处理的任务，比如，文件操作，需要先打开文件，操作完成后需要关闭文件。如果不使用with，文件操作通常得这样：

fp = open(r"D:\CSDN\Column\temp\mpmap.py", 'r')

try:

contents = fp.readlines()

finally:

fp.close()

如果使用 with - as，那就优雅多了：

>>> with open(r"D:\CSDN\Column\temp\mpmap.py", 'r') as fp:

contents = fp.readlines()

**5. 列表推导式**

在各种稀奇古怪的语法中，列表推导式的使用频率应该时最高的，对于代码的简化效果也非常明显。比如，求列表各元素的平方，通常应该这样写（当然也有其他写法，比如使用map函数）：

>>> a = [1, 2, 3, 4, 5]

>>> result = list()

>>> for i in a:

result.append(i\*i)

>>> result

[1, 4, 9, 16, 25]

如果使用列表推导式，看起来就舒服多了：

>>> a = [1, 2, 3, 4, 5]

>>> result = [i\*i for i in a]

>>> result

[1, 4, 9, 16, 25]

事实上，推导式不仅支持列表，也支持字典、集合、元组等对象。有兴趣的话，可以自行研究。我有一篇博文[《一行 Python 代码能实现什么丧心病狂的功能？》](https://blog.csdn.net/xufive/article/details/96475103)，里面的例子，都是列表推导式实现的。

**6. 列表索引的各种骚操作**

Python 引入负整数作为数组的索引，这绝对是喜大普奔之举。想想看，在C/C++中，想要数组最后一个元素，得先取得数组长度，减一之后做索引，严重影响了思维的 连贯性。Python语言之所以获得成功，我个人觉得，在诸多因素里面，列表操作的便捷性是不容忽视的一点。请看：

>>> a = [0, 1, 2, 3, 4, 5]

>>> a[2:4]

[2, 3]

>>> a[3:]

[3, 4, 5]

>>> a[1:]

[1, 2, 3, 4, 5]

>>> a[:]

[0, 1, 2, 3, 4, 5]

>>> a[::2]

[0, 2, 4]

>>> a[1::2]

[1, 3, 5]

>>> a[-1]

5

>>> a[-2]

4

>>> a[1:-1]

[1, 2, 3, 4]

>>> a[::-1]

[5, 4, 3, 2, 1, 0]

如果说，这些你都很熟悉，也经常用，那么接下来这个用法，你一定会感觉很神奇：

>>> a = [0, 1, 2, 3, 4, 5]

>>> b = ['a', 'b']

>>> a[2:2] = b

>>> a

[0, 1, 'a', 'b', 2, 3, 4, 5]

>>> a[3:6] = b

>>> a

[0, 1, 'a', 'a', 'b', 4, 5]

**7. lambda函数**

lambda 听起来很高大上，其实就是匿名函数（了解js的同学一定很熟悉匿名函数）。匿名函数的应用场景是什么呢？就是仅在定义匿名函数的地方使用这个函数，其他地 方用不到，所以就不需要给它取个阿猫阿狗之类的名字了。下面是一个求和的匿名函数，输入参数有两个，x和y，函数体就是x+y，省略了return关键字。

>>> lambda x,y: x+y

<function <lambda> at 0x000001B2DE5BD598>

>>> (lambda x,y: x+y)(3,4) # 因为匿名函数没有名字，使用的时候要用括号把它包起来

匿名函数一般不会单独使用，而是配合其他方法，为其他方法提供内置的算法或判断条件。比如，使用排序函数sorted对多维数组或者字典排序时，就可以指定排序规则。

>>> a = [{'name':'B', 'age':50}, {'name':'A', 'age':30}, {'name':'C', 'age':40}]

>>> sorted(a, key=lambda x:x['name']) # 按姓名排序

[{'name': 'A', 'age': 30}, {'name': 'B', 'age': 50}, {'name': 'C', 'age': 40}]

>>> sorted(a, key=lambda x:x['age']) # 按年龄排序

[{'name': 'A', 'age': 30}, {'name': 'C', 'age': 40}, {'name': 'B', 'age': 50}]

再举一个数组元素求平方的例子，这次用map函数：

>>> a = [1,2,3]

>>> for item in map(lambda x:x\*x, a):

print(item, end=', ')

1, 4, 9,

**8. yield 以及生成器和迭代器**

yield 这词儿，真不好翻译，翻词典也没用。我干脆就读作“一爱得”，算是外来词汇吧。要理解 yield，得先了解 generator（生成器）。要了解generator，得先知道 iterator（迭代器）。哈哈哈，绕晕了吧？算了，我还是说白话吧。

话说py2时代，range()返回的是list，但如果range(10000000)的话，会消耗大量内存资源，所以，py2又搞了一个 xrange()来解决这个问题。py3则只保留了xrange()，但写作range()。xrange()返回的就是一个迭代器，它可以像list那 样被遍历，但又不占用多少内存。generator（生成器）是一种特殊的迭代器，只能被遍历一次，遍历结束，就自动消失了。总之，不管是迭代器还是生成 器，都是为了避免使用list，从而节省内存。那么，如何得到迭代器和生成器呢？

python内置了迭代函数 iter，用于生成迭代器，用法如下：

>>> a = [1,2,3]

>>> a\_iter = iter(a)

>>> a\_iter

<list\_iterator object at 0x000001B2DE434BA8>

>>> for i in a\_iter:

print(i, end=', ')

1, 2, 3,

yield 则是用于构造生成器的。比如，我们要写一个函数，返回从0到某正整数的所有整数的平方，传统的代码写法是这样的：

>>> def get\_square(n):

result = list()

for i in range(n):

result.append(pow(i,2))

return result

>>> print(get\_square(5))

[0, 1, 4, 9, 16]

但是如果计算1亿以内的所有整数的平方，这个函数的内存开销会非常大，这是 yield 就可以大显身手了：

>>> def get\_square(n):

for i in range(n):

yield(pow(i,2))

>>> a = get\_square(5)

>>> a

<generator object get\_square at 0x000001B2DE5CACF0>

>>> for i in a:

print(i, end=', ')

0, 1, 4, 9, 16,

如果再次遍历，则不会有输出了。

**9. 装饰器**

刚弄明白迭代器和生成器，这又来个装饰器，Python 咋这么多器呢？的确，Python 为我们提供了很多的武器，装饰器就是最有力的武器之一。装饰器很强大，我在这里尝试从需求的角度，用一个简单的例子，说明装饰器的使用方法和制造工艺。

假如我们需要定义很多个函数，在每个函数运行的时候要显示这个函数的运行时长，解决方案有很多。比如，可以在调用每个函数之前读一下时间戳，每个函 数运行结束后再读一下时间戳，求差即可；也可以在每个函数体内的开始和结束位置上读时间戳，最后求差。不过，这两个方法，都没有使用装饰器那么简单、优 雅。下面的例子，很好地展示了这一点。

>>> import time

>>> def timer(func):

def wrapper(\*args,\*\*kwds):

t0 = time.time()

func(\*args,\*\*kwds)

t1 = time.time()

print('耗时%0.3f'%(t1-t0,))

return wrapper

>>> @timer

def do\_something(delay):

print('函数do\_something开始')

time.sleep(delay)

print('函数do\_something结束')

>>> do\_something(3)

函数do\_something开始

函数do\_something结束

耗时3.077

timer() 是我们定义的装饰器函数，使用@把它附加在任何一个函数（比如do\_something）定义之前，就等于把新定义的函数，当成了装饰器函数的输入参数。 运行 do\_something() 函数，可以理解为执行了timer(do\_something) 。细节虽然复杂，不过这么理解不会偏差太大，且更易于把握装饰器的制造和使用。

**10. 巧用断言assert**

所谓断言，就是声明表达式的布尔值必须为真的判定，否则将触发 AssertionError 异常。严格来讲，assert是调试手段，不宜使用在生产环境中，但这不影响我们用断言来实现一些特定功能，比如，输入参数的格式、类型验证等。

>>> def i\_want\_to\_sleep(delay):

assert(isinstance(delay, (int,float))), '函数参数必须为整数或浮点数'

print('开始睡觉')

time.sleep(delay)

print('睡醒了')

>>> i\_want\_to\_sleep(1.1)

开始睡觉

睡醒了

>>> i\_want\_to\_sleep(2)

开始睡觉

睡醒了

>>> i\_want\_to\_sleep('2')

Traceback (most recent call last):

File "<pyshell#247>", line 1, in <module>

i\_want\_to\_sleep('2')

File "<pyshell#244>", line 2, in i\_want\_to\_sleep

assert(isinstance(delay, (int,float))), '函数参数必须为整数或浮点数'

AssertionError: 函数参数必须为整数或浮点数