

python3_notes

- 数和表达式

// 数学运算舍弃小数部分，即执行整除运算 $1//2 = 0$

% 求余(求模)运算, $x \% y$ 结果为 x 除以 y 的余数: $x \% y$ 等价于 $x - ((x//y)*y)$

`pow(x,y)` 求幂, 等同于 $x^{**}y$

`abs` 求绝对值

`int` 取整数

`round` 将浮点数圆整为最接近的整数

- 变量

在python中, 名称(标示符)只能由字母, 数字和下划线构成, 且不能以数字打头

- 获得用户输入

单语句块, 指仅包含一句的语句块, 可以在同一行指定, 例如条件语句和循环

```
if 1==2:print('One equals two')
```

- 注释

井号(#), 开启解释行。程序员应准守, 勤注释, 注释务必言而有物, 不要重复去讲通过代码很容易获得的信息, 无用的注释还不如没有。

- 单双引号字符串以及对引号转义

若字符串中有单引号, 应使用双引号, 避免混淆; 若字符串包含双引号, 那么使用单引号将整个字符串括起。也可以使用转义符反斜杠(\)转移。

- 拼接字符串

可依次输入两个字符串, python则自动将其拼接一起; 而输入两个变量时则需使用加号(+): `x="hello,"`
`"; y="world!" ; x+y` `'hello, world!'`

- 字符串表示str和repr

python打印(`print`)的所有字符串, 都用引号括起, 不管单双引号都会对符号转移打印。用户可通过两种不同机制将值转换为字符串,str和repr。

`str`能以合理的方式将值转换为用户能够看懂的字符串, 尽可能将特殊字符编码转换为相应的字符

`repr`则获得值的合法python表达式表示

```
print(repr('hello, \nworld!')); 'hello, \n world!'
```

```
print(str('hello, \nworld!'));
```

```
hello,
```

```
world!
```

- 长字符串，原始字符串和字节

跨越多行的字符串，可使用三引号(而不是普通引号,""或"""")

```
print9'''This is a very long string....''')
```

或者简单使用反斜杠(\)转移换行

使用反斜杠(\)进行特殊字符的原始字符输出: \\n；或者使用原始字符串前缀r函数，输出原始字符串，此时，原始字符串不能以单个反斜杠结尾。

```
print(r'let\s go!'); let'\s go!
```

- 序列

通用序列操作包含索引，切片，相加，相乘和成员资格检查；同时python提供了一些内置函数，用于确定序列的长度以及查找序列中的最大，最小元素

序列中所有元素都有编号，从0开始从左到右递增；相反从右到左时，从-1开始索引

使用切片(slicing)访问特定范围内的元素tag[9:30:1]，其中第一数字为起点，第二为终点，第三个为步长；同时遵守左开右合的规范，同样适用于负数索引。tag[-3:]为最后三个元素;tag[:3]为开头三个元素;tag[:]为整个序列; tag[::-1]表示反转tag字符串，步长为负表示从右往左提取元素

序列相加时要针对同类型序列

序列与数字相乘时，表示重复序列

空表格为[]，若创建未添加任何内容的10个元素的表格: [None]*10，使用None

使用运算符in检测序列中是否包含特定的值，满足返回True，不满足返回False: 'w' in 'world'; True

len(numbers)，返回元素个数；max(numbers)，返回最大值；min(numbers)，返回最小值；注意max和min要strings/numbers中类型相同

- 列表

使用函数list创建列表

```
list('hello')
```

```
['h', 'e', 'l', 'l', 'o']
```

如要将字符列表转换为字符串，使用join: ''.join(somelist)

可对列表执行所有标准序列操作，如索引，切片，拼接和相乘等

可使用索引对列表修改或者给元素赋值，但是不能给不存在(索引)的列表元素赋值

删除元素，del语句即可：del names[2]，同时列表长度缩短，也可使用del删除字典和变量

使用切片对列表赋值，替换值，或者替换为长度与其不同的序列

可在不替换原有元素的情况下插入新元素: numbers=[1,5]; numbers[1:1]=[2,3,4]; numbers=[1,2,3,4,5]；也可以插入空序列来删除切片 numbers=[]，等效于 del numbers[:]

方法是与对象(列表, 数, 字符串等)联系紧密的函数: `object.method(arguments)`

`append` 将一个对象附加到列表末尾, 直接修改旧列表而不是返回新列表: `lst=[1,2,3]; list.append(4)`

`clear` 就地清空列表内容: `lst.clear()`, 类似于: `lst[:] = lst[]`

`copy` 复制列表, 常规复制(等号赋值方式)只是将一个名称关联到列表: `b=a.copy()`

`count` 计算指定元素在列表中出现次数: `x.count(1)`, 注意列表中含有的列表一起成为一个元素

`extend` 同时将多个值附加到列表末尾, `append`仅增加一个元素到末尾: `a.extend(b)`, 而采用加号(+)会返回一个新的列表, `extend`为直接修改原列表; 或采用切片方式直接赋值: `a[len(a):]=b`

`index` 在列表中查找指定值第一次出现的索引: `knights.index('who')`

`insert` 用于将一个对象插入列表: `numbers.insert(3, 'four')`, 在第3位置插入'four'; 或采用切片方式: `numbers[3:3]=['four']`

`pop` 从列表中删除一个元素(末尾为最后一个元素), 并返回这一元素: `x.pop(0)`, 默认删除最后一个元素。`pop`是唯一一个即修改列表又返回一个非None值的列表方法

栈(stack), 常见数据结构有先进先出(FIFO)或后进先出(LIFO)两种操作: `x.append(x.pop()); x.insert(0,x.pop())`

`remove` 删除第一个为指定值的元素: `x.remove('be')`, 就地删除

`reverse` 按照相反顺序排列列表中的元素: `x.reverse()`, 就地修改

`sort` 对列表就地排序: `x.sort(); y=x.sort() # Dont do this!`

`sorted` 获得排序后列表的副本: `y=sorted(x)`, `y`等与 `x.sort()`; 列表x顺序不变, 该函数可适用于任何序列, 但总是返回一个列表: `sorted('Python'), ['P', 'h', 'n', 'o', 't', 'y']`

`sort` 高级排序, 可接受两个可选参数: `key, reverse`, 参数`key`类似于参数`cmp`: 设置为一个用于排序的函数, 用该函数为每个元素创建一个键, 再根据键对元素进行排序, 例如根据元素长度排序:

`x.sort(key=len);` 而`reverse`, 只需指定为—True或False, 表明是否按照相反顺序排序:

`x.sort(reverse=True); sorted`也可接受参数`key`和`reverse`

- 元组: 不可修改的序列

元组语法简单, 只需将一些值逗号分隔就自动创建一个元组: `>>> 1,2,3; (1,2,3)`

通常使用圆括号括起: `>>> (1,2,3) ; (1,2,3)`; 空元组使用不包含任何内容的圆括号表示: `()`; 仅包含一个值的元组: `42, ; (42,)`

`3 (40+2); 126; 3 (40+2,); (42,42,42)`

`tuple`工作原理与`list`很像: 将一个序列作为参数, 并转为元组, 若参数本身就是元组, 则原封不动返回: `tuple([1,2,3]); (1,2,3); tuple('abc'); ('a','b','c'); tuple((1,2,3)); (1,2,3)`

元组也可采用切片方法, 返回仍然是元组, 一般列表足以满足对序列的需求。

- 设置字符串格式

对字符串调用format方法，并提供要设置其格式的值，每个值都被插入到字符串中，以替换用花括号括起来的替换字段，要在作用结果中包含花括号，可在格式字符中使用两个花括号{{}}来指定：{替换部分}.format(格式部分)

按顺序将字段和参数配对，还可以给参数指定名称：“{foo} {} {bar}
{}`.format(1,2,bar=4,foo=3)：'3 1 4 2'

通过索引指定要在哪个字段中使用相应的未命名参数，可不按顺序使用未命名参数：“{foo} {1} {bar}
{0}`.format(1,2,bar=4,foo=3)：'3 2 4 1'

或采用访问组成部分的方法：“fullname=['Alfred', 'Smoketoomuch']："Mr
{name[1]}".format(name=fullname) Mr Smoketoomuch

使用转换标志设置格式指定：“print ("{pi:s} {pi:r} {pi:a}" .format(pi="\$\pi\$")); \$\pi\$
'\$\pi\$' '\u03c0'

s, r, a分别使用了str, repr和ascii进行转换；函数str创建外观普通的字符串函数；repr函数创建给定值的python表示，这里是一个字符串的字面量；ascii创建只包含ASCII字符的表示。

指定转换值的类型，在冒号后加指定类型说明符：f浮点型；d十进制整数；e科学记数法表示小数（e表示指数）；%将数表示为百分比值（乘以100，在后面加上%）

设置浮点型时，默认在小数点后面显示6位小数，并根据需要设置字符的宽度，而不进行任何形式的填充：“
{num:10}`.format(num=3)，'. 3'

"{name:10}`.format(name="Bob")，'Bob' 数和字符串的对齐方式不同

精度的指定需要在它前面加上一个表示小数点的句号：“Pi day is {pi:.2f}`.format(pi=pi)，'Pi
day is 3.14'

使用逗号指出要添加千位分割符：“One googol is {:.,.}`.format(10**100)"，'One googol is
10,000,...'；同时设置其他格式时，逗号应放在宽度和表示精度的句点之间

在指定宽度和精度的数前面，可添加一个标志，可以是零、加号、减号或空格，其中零表示使用0来填充数字：“{:010.2f}`.format(pi)，'0000003.14'

使用<,>和^表示左对齐、右对齐和居中：“{:<15}`.format(" WIN BIG ")，'\$\$\$ WIN BIG \$\$\$'，总共为15个字符长度

更具体的说明符=，它将指定填充字符放在符号和数字之间

若给正数加上符号，可使用说明符+，将其放在对齐说明符后面：

print('{0:+.2}\n{1:+.2}`.format(pi,-pi))，'+3.1，-3.1

- 字符串方法

模块string，包含了一些字符串没有的常量和函数：string.digits，包含了数字0~9的字符串；string.ascii_letters，包含了所有ASCII字母（大写和小写）的字符串；string.ascii_lowercase，包含了所有小写ASCII字符的字符串；string.printable，包含所有可以打印的ASCII字符的字符串

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
!#$%&\ '()*+, -./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

string.punctuation, 包含所有ASCII标点字符的字符串

```
>>> string.punctuation  
'!"#$%&\ '()*+, -./:; <=>?@[\\]^_`{|}~'
```

方法center通过在两边添加填充字符(默认为空格)让字符串剧中: "The Middle by Jimmy Eat World".format(39, "*"), *****The Middle by Jimmy Eat World*****

方法find在字符串中查找子串, 返回第一个字符的索引, 否则返回-1: title.find('Monty')

同时可以指定搜索的起点和终点(可选选项): subject.find('\$\$\$', 1, 16), 对应起点和终点

join方法可以合并序列元素: '+' .join(seq), 所有合并序列的元素必须都是字符串, 不能为数字:

```
'+'.join(seq=['1', '2', '3', '4'])
```

```
>>> dirs='','usr','bin','env'  
>>> dirs  
('', 'usr', 'bin', 'env')  
>>> ''.join(dirs)  
'usrbinenv'  
>>> '/'.join(dirs)  
'/usr/bin/env'
```

lower方法 返回字符串的小写版本; title方法返回字符串为词首大写, 或使用string中的capwords, 将词首转换为大写; upper方法返回字符串的大写版本

replace方法将指定子串都替换为另一个字符串, 并返回替换后的结果: seq.replace('is', 'eez')

split方法将字符串拆分为序列: '1+2+3+4+5'.split('+'), 若没有指定分隔符, 将默认在单个或多个连续的空白字符(空格, 制表符, 换行符等)处进行拆分, 返回list

strip方法将字符串开头和末尾的空白(但不包含中间的空白)删除, 并返回删除后的结果: seq.strip()

同时还可以在一个字符串参数中指定要删除哪些字符: '*** SPAM for everyone!!! ***'.strip('*!'), 'SPAM for everyone' 该方法仅删除开头或末尾的指定字符

translate方法和replace一样替换字符串的特定部分, 但是不同的是它只能进行单字符替换, 该方法可以同时替换多个字符, 然而使用前必须创建一个转换表, 该表指出不同Unicode码点之间的转换关系

使用is开头的字符串方法用于判断字符串是否具有特定性质: isalnum, isalpha, isdecimal, isdigit, isidentifier, isspace, isupper

string.capwords(s[,seq]), 使用split根据sep拆分s, 将每项的首字母大写, 再以空格为分隔符将它们合并起来

ascii(obj)创建指定对象的ASCII表示

- 字典

该数据结构称为映射(mapping), 字典是python中唯一的内置映射类型, 其中的值不按顺序排列, 而是存储在键下, 键可能是数, 字符串或元组

字典由键及相应的值组成, 这种键-值对称为项(item), 每个键与其值之间都用冒号(:)分隔, 项之间用逗号分隔, 而整个字典都放在花括号内; 空字典(没有任何项)用两个花括号表示, 类似于:{}; 字典中键必须是独一无二的, 而字典中的值无需如此

函数dict可从其他映射(其他字典)或键-值对序列创建字典, 同list, tuple, str一样, dict不是函数, 而是一个类:`dict([('name', 'Gumby'), ('age', 42)])`

`len(d)`返回字典d中的项(键-值对)数

`d[k]`返回与键k相关联的值

`d[k]=v`将值v关联到键k

`del d[k]`删除键为k的项

`k in d`检查字典d是否包含键为k的项, 查找的是键而不是值; 而表达式`v in l`(l是一个列表)查找的是值而不是索引

将字符串格式设置功能用于字典, 可在字典中包含各种信息, 只需在格式字符串中提取所需的信息即可, 必须使用format_map来指处你将通过一个映射来提供所需的信息:`"Cecil's phone number is {Cecil}."`.format_map(phonebook), phonebook为字典: `{'Beth': '9102', 'Alice': '2341', 'Cecil': '3258'}`

`clear`方法删除所有的字典项, 该操作就像list.sort一样, 就地执行: `d.clear()`, d {}

`copy`方法返回一个新字典, 其中包含的键-值对与原来的字典相同, 为浅复制, 当替换副本中值时, 原件不受影响, 然而修改副本中的值, 原件也将发生变化, 因为原件指向的也就是被修改的值

可使用copy中的deepcopy函数, 执行深度复制: `dc.deepcopy(d)`

`fromkeys`方法创建一个新字典, 其中包含指定的键, 且每个键对应的值都是None:

`'{} .fromkeys(['name', 'age'])'`, `{'age': None, 'name': None}`

或 `dict.fromkeys(['name', 'age'])`

或使用特定值: `dict.fromkeys(['name', 'age'], '(unknown)')`

`get`方法用于访问字典, 一般试图访问字典中没有的项, 将引发错误, 而使用get访问不存在的键时, 不会引发异常, 而是返回None: `print(d.get('name'))`

或指定不存在键时, 返回特定值: `d.get('name', 'N/A')`

`items`方法返回一个包含所有字典项的列表, 其中每个元素都为(key, value)的形式, 返回值属于一种名为字典视图的特殊类型, 字典视图可用于迭代

或将字典项复制到列表中: `list(d.items())`

视图的一个优点是不复制, 它们始终是底层字典的反映, 即使你修改了底层字典亦是如此

```
>>> ('title', 'Pyhton Web Site') in d.items()
True
>>> d
{'title': 'Pyhton Web Site', 'Spam': 0}
>>> ('title', 'Pyhton Web Site') in d.items()
```

`keys`方法返回一个字典视图, 其中包含指定字典中的键

`values`方法返回一个由字典中的值组成的字典视图, 不同与keys, values返回的视图可能包含重复的值

`pop`方法用于获取与指定键相关联的值, 并将该键-值对从字典中删除

`popitem`方法类似于`list.pop`, `list.pop`弹出列表中的最后一个元素, 而`popitem`随机地弹出一个字典项, 因为字典项的顺序是不确定的, 没有'最后一个元素'的概念

`setdefault`方法类似`get`, 也获得与指定键相关联的值, 此外, `setdefault`还在字典不包含指定的键时, 在字典中添加指定的键-值对: `d={}`, `d.setdefault('name', 'N/A')`, `'N/A'`

`update`方法使用字典中的项来更新另一个字典: `d.update(x)`, 通过参数提供的字典, 将其项添加到当前字典中, 若当前字典包含键相同的项, 就替换它

- import及赋值

导入函数别名: `from math import sort as foobar`, 本地重命名函数

并行给多个变量赋值: `x,y,z=1,2,3`, `print(x,y,z)`, `1 2 3`

使用星号*来收集多余的值, 确保值和变量的个数相同: `a,b,*rest=[1,2,3,4]`, `rest=[3,4]`, 也可将星号*放到其他位置收集, *指定的变量为列表

链式复制: `x=y=somefunction()`

增强复制: `x+=1`, `x*=2`, 也可针对字符串

- 条件和条件语句

条件语句视为假: `False None 0 '' () [] {}`, 其他值都为真, 包括特殊值`True`

```
>>> True
True
>>> False
False
>>> True == 1
True
>>> False == 0
True
```

条件表达式: `status='friend' if name.endswith('Gumy') else 'stranger'`, 若为真提供`friend`, 若为假提供`stranger`

`elif`用于`if`循环检查多个条件

python比较运算符

表达式	描述
<code>x == y</code>	x等于y
<code>x < y</code>	x小于y
<code>x > y</code>	x大于y
<code>x >= y</code>	x大于或等于y
<code>x <= y</code>	x小于或等于y
<code>x != y</code>	x不等于y
<code>x is y</code>	x和y是同一个对象
<code>x is not y</code>	x和y是不同的对象
<code>x in y</code>	x是容器(如序列)y的成员
<code>x not in y</code>	x不是容器(如序列)y的成员

is, 相同运算符, 看似与==一样, 不同在于检查两个对象是否相同, 而不是相等

运算符in用于成员资格检查

字符串的比较时根据字符的字母排列顺序进行比较的

```
>>> ord("߱")
128585
>>> ord("߲")
128586
>>> chr(128584)
߱'
```

and布尔运算符接受两个真值, 且都为真时返回真; or表满足一个即可; not表示非

asser断言, 可在不满足条件时退出: `assert 0 < age < 100`, 充当程序的检查点

也可在条件后面添加一个字符串, 对断言做出说明: `assert 0 < age < 100, 'The age must be realistic'`

- 循环

`while/for: while x < =100:print(x), x+=1, for word in words:print(word)`

range函数创建范围的内置函数: `list(range(0,10))`, 范围类似切片, 包含其实位置(这里是0), 但是不包含结束位置(这里为10): `[0,1,2,3,4,5,6,7,8,9]`

遍历字典所有关键字: `for key in d:print(key, 'corresponds to', d[key])`

或使用序列解包: `for key, value in d.items():print(key, 'corresponds to', value)`

- 迭代

函数zip将连个序列'缝合'起来, 并返回一个由元组组成的序列, 返回值是一个适合迭代的对象, 要查看其内容, 可使用list将其转换为列表:

```
names = ['anne', 'beth', 'george', 'damon']
ages = [12, 45, 32, 102]
```

zip缝合后可在循环中将元组解包:

```
>>> list(zip(names, ages))
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
>>> for name, age in zip(names, ages):
...     print(name, 'is', age, 'years old')
...
anne is 12 years old
beth is 45 years old
george is 32 years old
damon is 102 years old
```

函数zip可用于'缝合'任意数量的序列, 当序列长度不一致时, 函数zip将在最短的序列用完后停止'缝合'

```
>>> list(zip(range(5), range(100)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

使用index一次查询字符'xxx'位置且替换:

```
index = 0
for string in strings:
    if 'xxx' in string:
        strings[index] = '[censored]'
    index += 1
```

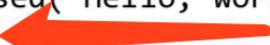
或使用内置函数enumerate:

```
for index, string in enumerate(strings):
    if 'xxx' in string:
        strings[index] = '[censored]'
```

enumerate能够迭代返回索引-值对, 其中索引是自动提供的

reversed/sorted, 不就地修改对象, 而是返回反排和排序后的版本, sorted返回一个列表, 而reversed像zip那样返回一个可迭代对象, 若使用索引或切片操作, 需先使用list对返回对象转换

```
>>> sorted([4, 3, 6, 8, 3])
[3, 3, 4, 6, 8]
>>> sorted('Hello, world!')
[' ', '!', ',', '.', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
>>> list(reversed('Hello, world!'))
['!', 'd', 'l', 'r', 'o', 'w', ' ', ' ', 'o', 'l', 'l', 'e', 'H']
>>> ''.join(reversed('Hello, world!'))
'!dlrow ,olleH'
```



例如:

```
>>> for index, value in enumerate(''.join(reversed('Hello, world!'))):
...     print('index', index, 'value', value)
...
index 0 value !
index 1 value d
index 2 value l
index 3 value r
index 4 value o
index 5 value w
index 6 value
index 7 value ,
index 8 value o
index 9 value l
index 10 value l
index 11 value e
index 12 value H
```

要对字母排序, 可先转换为小写, 为此可讲sorted的key参数设置为str.lower: `sorted('aBc', key=str.lower)`

- 跳出循环

`break`结束(跳出)循环; `continue`结束当前迭代, 并跳到下一次迭代开头, 这意味着跳过循环体中余下的语句, 但是不结束循环

```
for x in seq:
    if condition1: continue
    if condition2: continue
    if condition3: continue

    do_something()
    do_something_else()
    do_another_thing()
    etc()
```

或:

```
for x in seq:
    if not (condition1 or condition2 or condition3):
        do_something()
        do_something_else()
        do_another_thing()
        etc()
```

- 简单推导

列表推导是一种从其他列表创建列表的方式, 类似于数学中的集合推导, 工作原理简单, 类似于for循环:

```
[x*x for x in range(10)], [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

或加入条件判断: `[x*x for x in range(10) if x % 3 == 0], [0, 9, 36, 81]`

或更多for部分:

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

等同于两个for循环构建同样列表:

```
result = []
for x in range(3):
    for y in range(3)
        result.append((x, y))
```

构建男孩/女孩名称配对:

```
girls = ['alice', 'bernice', 'clarice']
boys = ['chris', 'arnold', 'bob']
letterGirls = {}
for girl in girls:
    letterGirls.setdefault(girl[0], []).append(girl)
print([b+'+'+g for b in boys for g in letterGirls[b[0]]])
```

这里使用圆括号代替方括号并不能实现元组推导

使用花括号执行字典推导:

```
>>> squares = {i:"{} squared is {}".format(i, i**2) for i in range(10)}
>>> squares[8]
'8 squared is 64'
```

pass语句什么也不做;del删除语句

exec将字符串作为代码执行, 加入命名空间:

```
>>> from math import sqrt
>>> scope = {}
>>> exec('sqrt = 1', scope)
>>> sqrt(4)
2.0
>>> scope['sqrt']
1
```

chr(n), 返回一个字符串, 其中只包含一个字符, 这个字符对应于传入的顺序值n(0<=n<=256)

ord(c), 接受一个只包含一个字符的字符串, 并返回这个字符的顺序值(一个整数)

range([start],stop[,step]), 创建一个用于迭代的range对象list(range())

zip(seq1,seq2,...), 创建一个使用于并行迭代的新序列

- 计算斐波那契数列

```
fibs=[0,1]

for i in range(8):

    fibs.append(fibs[-2]+fibs[-1])
```

- 自定义函数

内置函数callable, 判断否个对象是否可调用

在def语句后面(以及模块和类的开头)添加注释字符串, 放在函数开头的字符串称为文档字符串, docstring, 将作为函数的一部分储存起来:

```
>>> def square(x):
...     'Calculates the square of the number x.'
...     return x*x
...
>>> square.__doc__
'Calculates the square of the number x.'
```

内置函数help可获得有关函数的信息:

```
Help on function square in module __main__:

square(x)
    Calculates the square of the number x.
(END)
```

函数中return后可不接内容, 表示结束函数; 函数内部给参数赋值对外部没有任何影响, 函数存在局部作用域中

字符串(以及数和元组)是不可变的(immutable), 这意味着不可能修改它们(即只能替换为新值); 而对于可变的数据结构(如列表), 会存在修改了变量关联到的列表, 因此会改变对应关联列表的值

```
>>> names=['Mrs. Entity', 'Mrs. Thing']
>>> n=names
>>> n[0]='Mr. Gumby'
>>> names
['Mr. Gumby', 'Mrs. Thing']
```

将同一个列表赋给两个变量时, 这两个变量将同时指向这个列表; 若要避免这样结果, 必须创建列表的副本, 对序列执行切片操作时, 返回的切片都是副本

函数参数赋值时指定参数的名称以便忽略调用时根据参数位置调用, 此类关键字参数最大优点在于可以指定默认值

```
hello_1(name='world', greeting='Hello')
```

- 收集参数

参数前带一星号表示收集余下的位置参数为元组: `def print_params(title, *params)`; 若没有可供收集的参数, params将是一个空元组

可使用两个星号收集关键字参数: `def print_params(**params)`; 这样得到的是一个字典而不是元组, 此时参数为字典形式: `x=1, y=2, c=3...`

- 分配参数

元组:

```
>>> def add(x,y):
...     return x+y
...
>>> params=(1,2)
>>> add(*params)
3
```

字典:

```
>>> def hello_3(name="world",greeting="Hello"):
...     print(greeting,name)
...
>>> hello_3()
Hello world
>>> params={'name':'Sir Robin','greeting':'Well met'}
>>> hello_3(**params)
Well met Sir Robin
```

- 作用域

变量为指向值的名称, 执行赋值语句x=1后, 名称x指向值1, 这几乎与使用字典时一样(字典中的键指向值), 内置函数vars可返回这个不可见的字典

可在函数中读取全局变量的值, 但是不能修改函数作用域外的全局变量值; 若有一局部变量或参数与想要访问的全局变量同名, 就无法访问全局变量了, 因为它被局部变量遮住了; 可使用函数globals来访问全局变量, 该函数类似于vars, 返回一个包含全局变量的字典, 而locals返回一个包含局部变量的字典

在函数中访问同名全局变量: `def combine(paramter): print(parameter + globals()['paramter'])`

可在函数内部给变量赋值时指明该默认局部变量为全局变量: `def change_global():global x`

- 作用域嵌套

python函数可以嵌套, 即将一个函数放在另一个函数内, 例如一个函数位于另一函数中, 且外面函数返回里面的函数:

```
>>> def multiplier(factor):
...     def multiplyByFactor(number):
...         return number*factor
...     return multiplyByFactor
...
>>> double=multiplier(2)
>>> double(5)
10
```

此类multiplyByFactor这样存储其所在的作用域的函数称为闭包, 通常不能给外部作用域内的变量赋值, 但如果一定要这样做, 可能使用关键字nonlocal, 能够给外部作用域(非全局作用域)内的变量赋值

- 递归

递归函数包含: 基线条件(针对最小的问题), 满足这种条件时函数将直接返回一个值; 递归条件: 包含一个或多个调用, 这些调用旨在解决问题的一部分. 问题最终被分解成基线条件可以解决的最小问题

阶乘:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

二分查找:

```
def search(sequence, number, lower, upper):
    if lower == upper:
        assert number == sequence[upper]
        return upper
    else:
        middle = (lower + upper) // 2
        if number > sequence[middle]:
            return search(sequence, number, middle + 1, upper)
        else:
            return search(sequence, number, lower, middle)
```

模块**bisect**提供了标准的二分查找实现

- 列表推导式

函数map使用每次迭代的参数计算函数: `list(map(str,range(10)))` 等同于 `list(str(i) for i in range(10))`

函数filter根据布尔函数的返回值对元素进行过滤:

```
>>> def func(x):
...     return x.isalnum()
...
>>> seq = ["foo", "x41", "?!", "***"]
>>> list(filter(func, seq))
['foo', 'x41']
```

等同于 `[x for x in seq if x.isalnum()]`

lambda表达式用于创建内嵌的简单函数(主要供map,filter使用): `filter(lambda x:x.isalnum(),seq)`

reduce函数针对序列项采用累增的方式, 从左到右依次减少序列直至单个值

`reduce(lambda x,y:x+y, [1,2,3,4,5])` 等同 `((((1+2)+3)+4)+5)`

```
>>> numbers = [72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108,
100, 33]
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, numbers)
1161
```

lambda源于希腊字母, 在数学中用于表示匿名函数

`apply(func[,args[,kwargs]])`: 调用函数(还提供要传递给函数的参数)

- 创建对象类型或类

与对象属性相关联的函数称为方法: `'abc'.count('a')`, 1

多态形式式python编程方式的和兴, 有时称为鸭子类型. 该术语源自如下说法: '如果走起来像鸭子, 叫起来像鸭子, 那么它就是鸭子'

- 封装(encapsulation)

封装指的是向外部隐藏不必要的细节, 不同与多态, 多态让你无需知道对象所属的类(对象的类型)就能调用其方法, 而封装让你无需知道对象的构造就能使用它(通过它的方法)

- 继承

类是一种对象, 每个对象都属于特定的类, 并被称为该类的实例, 例如云雀为鸟类的子类, 鸟类为云雀的超类; 英语日常交谈中, 使用复数来表示类, 如birds, larks(云雀), 在python中约定使用单数并将首字母大写, Birds/Lark, 表示类

类由其所支持的方法定义的, 类的所有实例(s=Filter())都有该类的所有方法, 因此子类的所有实例都有超类的所有方法, 要定义子类, 只需定义多出来的方法

- 常见自定义类

```
class Person:

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def greet(self):
        print("Hello, world! I'm {}".format(self.name))
```

该例子定义了三个方法, 类似函数定义. 参数self指定对象本身, self很有用, 甚至必不可少, 若是没有它, 所有方法都无法访问对象本身---要操作的属性所属的对象, 当然也无法从外部访问这些属性

方法和函数的区别表现在参数self上

若让类的方法或属性称为私有的(不能通过方法从外部访问), 只需让其名称以两个下划线开头即可

```
>>> class Secretive:
...     def __inaccessible(self):
...         print('Bet you cant see me...')
...     def accessible(self):
...         print('The secret message is:')
...         print('Bet you cant see me..')
...
```

这样, 从外部不能访问 `__inaccessible`, 但是在类中, `accesssible` 依然可以使用

对于私有方法/属性, 可以通过开头加一个下划线和类名的方式访问:

```
>>> s.accessible()
The secret message is:
Bet you cant see me..
>>> s._Secretive__inaccessible
<bound method Secretive._inaccessible of <__main__.Secretive object at 0x10a27ebd0>>
>>> s._Secretive__inaccessible()
Bet you cant see me...
```

在class语句中定义的代码都是在一个特殊的命名空间(类的命名空间)内执行的, 而类的所有成员都可以访问这个命名空间; 类定义其实就是要执行的代码段, 因此在类的定义中, 并非只能包含def语句

```
>>> class MemberCounter:
...     members=0
...     def init(self):
...         MemberCounter.members+=1
...
...
```

该类每次调用都会使用init来初始化所有实例

要指定超类, 可在class语句中的类名后加上超类名, 并将其用圆括号括起来

```
>>> class Filter:
...     def init(self):
...         self.blocked=[]
...     def filter(self,sequence):
...         return [x for x in sequence if x not in self.blocked]
...
>>> class SPAMFilter(Filter):
...     def init(self):
...         self.blocked=['SPAM']
...
>>> s=SPAMFilter()
>>> s.init()
>>> s.blocked
['SPAM']
```

重写了Filter类中的方法init的定义; 直接从Filter类继承了方法filter的定义, 因此无需重新编写其他定义

```
>>> s.filter(['SPAM','SPAM','eggs','bacon','SPAM'])
['eggs', 'bacon']
```

函数issubclass用于判断一个类是否是另一个类的子类: `issubclass(SPAMFilter, Filter)`

查看一个类是否含有基类, 访问其特殊属性`_base_`: `SPAMFilter.__base__`

```
>>> SPAMFilter.__base__
<class '__main__.Filter'>
```

使用函数isinstance判断对象是否是特定类的实例:

```
>>> s=SPAMFilter()
>>> isinstance(s,SPAMFilter)
True
>>> isinstance(s,Filter)
True
>>> isinstance(s,str)
False
```

查看对象所属的类, 可使用属性`__class__`: `s.__class__`

```
>>> s.__class__
<class '__main__.SPAMFilter'>
```

还可以使用`type(s)`获悉其所属的类: `type(s)`

- 多个超类

```
>>> class Calculator:
...     def calculate(self,expression):
...         self.value=eval(expression)
...
>>> class Talker:
...     def talk(self):
...         print("Hi, my value is",self.value)
...
>>> class TalkingCalculator(Calculator,Talker):
...     pass
...
>>> tc=TalkingCalculator()
>>> tc.calculate('1+2*3')
>>> tc.talk()
Hi, my value is 7
```

子类TalkingCalculator本身无所作为, 其所有的行为都是从超类继承的; 从Calculator继承了calculate, 并从Talker继承talk

多个超类使用复数形式的`__bases__`:

```
>>> TalkingCalculator.__base__
<class '__main__.Calculator'>
>>> TalkingCalculator.__bases__
(<class '__main__.Calculator'>, <class '__main__.Talker'>)
```

若多个超类以不同的方式实现了同一个方法(即有多个同名方法), 必须在`class`语句中小心排列这些超类, 因为位于前面的类的方法将覆盖位于后面的类的方法

- 接口和内省

接口这一概念与多态相关, 处理多态对象是, 只关心其接口(协议)-对外暴露的方法和属性; 在pyhton中, 不显示地指定对象必须包含哪些方法才能作用参数

`hasattr`函数用于检查所需方法是否存在: `hasattr(tc, 'talk')`

`callable/getattr`(指定属性不存在是使用默认值, 这里为None)函数用于检查属性是否可调用:
`callable(getattr(tc, 'talk', None))`

`setattr`与`getattr`功能相反, 可用于设置对象的属性:

```
>>> setattr(tc, 'name', 'Mr. Gumby')
>>> tc.name
'Mr. Gumby'
```

要查看对象中存储的所有值, 可检查其`__dict__`属性

```
>>> tc.__dict__
{'value': 7, 'name': 'Mr. Gumby'}
```

- 抽象基类

P229略

小结:

- 对象: 对象由属性和方法组成。属性不过是属于对象的变量, 而方法是存储在属性中的函数。相比于其他函数, (关联的)方法有一个不同之处, 那就是它总是将其所属的对象作为第一个参数, 而这个参数通常被命名为`self`。
- 类: 类表示一组(或一类)对象, 而每个对象都属于特定的类。类的主要任务是定义其实例将包含的方法。
- 多态: 多态指的是能够同样地对待不同类型和类的对象, 即无需知道对象属于哪个类就可调用其方法。
- 封装: 对象可能隐藏(封装)其内部状态。在有些语言中, 这意味着对象的状态(属性)只能通过其方法来访问。在Python中, 所有的属性都是公有的, 但直接访问对象的状态时程序员应谨慎行事, 因为这可能在不经意间导致状态不一致。
- 继承: 一个类可以是一个或多个类的子类, 在这种情况下, 子类将继承超类的所有方法。你可指定多个超类, 通过这样做可组合正交(独立且不相关)的功能。为此, 一种常见的做法是使用一个核心超类以及一个或多个混合超类。
- 接口和内省: 一般而言, 你无需过于深入地研究对象, 而只依赖于多态来调用所需的方法。然而, 如果要确定对象包含哪些方法或属性, 有一些函数可供你用来完成这种工作。
- 抽象基类: 使用模块`abc`可创建抽象基类。抽象基类用于指定子类必须提供哪些功能, 却不实现这些功能。
- 面向对象设计: 关于该如何进行面向对象设计以及是否该采用面向对象设计, 有很多不同的观点。无论你持什么样的观点, 都必须深入理解问题, 进而创建出易于理解的设计。

- 异常处理

事实上, 每个异常都是每个类的实例, 可通过各种方法引发和捕获这些实例, 从而逮住错误并采取措施, 而不是放任整个程序失败

`raise`语句将一个类(必须是`Exception`的子类)或实例作为参数; 将类作为参数时, 将自动创建一个实例
`raise Exception`通用异常; `raise Exception('hyperdrive overload')`指定错误消息'hyperdrive overload'

python库参考手册的'Built-in Exceptions', 描述了所有内置异常类, 都可用于raise语句: `raise ArithmeticError`

创建异常类, 务必直接或间接地继承Exception(这意味着从任何内置异常类派生都可以): `class SomeCustomException(Exception):pass`

- 捕获异常

使用try/except语句捕获异常

```
>>> try:  
...     x=int(input('Enter the first number: '))  
...     y=int(input('Enter the second number: '))  
...     print(x/y)  
... except ZeroDivisionError:  
...     print('The second number can't be zero!')
```

若异常捕获后, 要重新引发它(即继续向上传播), 可调用raise且不提供任何参数

如果无法处理异常, 在except子句中使用不带参数的raise通常是不错的选择, 但有时你可能想引发别的异常; 在这种情况下, 导致进入except子句的异常将作为异常上下文存储起来, 并出现在最终的错误消息中

可使用raise...from...语句来提供自己的异常上下文, 也可使用None来禁用上下文

```
>>> try:  
...     1/0  
... except ZeroDivisionError:  
...     raise ValueError from None  
...  
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
ValueError
```

可在try/except语句中添加多个except子句, 捕获多种错误, 或使用元组指定异常: `except (ZeroDivisionError, TypeError, NameError)`; 若except子句没有提供错误或异常的名称, 它将处理所有错误与异常

在没有异常时执行一个代码很有用, 为此, 可像条件语句和循环一样, 给try/except语句添加一个else子句

```
>>> try:  
...     print('A simple task')  
... except:  
...     print('What? something went wrong?')  
... else:  
...     print('Ah... It went as planned')  
...  
A simple task  
Ah... It went as planned
```

最后, 使用finally子句可用在发生异常时执行清理工作, 与try子句配套

```
>>> x=None
>>> try:
...     x=1/0
... finally:
...     print('Cleaning up ..')
...     del x
...
Cleaning up ..
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

因此, 不管try子句中发生了什么异常, 都将执行finally子句

异常处理并不是很复杂, 如果你知道代码可能引发某种异常, 且不希望出现这种异常时程序终止并显示栈跟踪消息, 可添加必要的try/except或try/finally语句(或结合使用)来处理它

关键是很多情况下, 相比于使用if/else, 使用try/except语句更自然, 也更符合python的风格, 因此你应该尽可能使用try/except语句的习惯, P253异常之蝉

python偏向使用try/except的原因, 这种策略可总结为习语'闭眼就跳'---直接去做, 有问题再处理, 而不是预先做大量的检查

- 魔法方法/特征/迭代器

创建构造函数, 只需将方法init的名称从普通的init改为魔法版__init__即可, 这样以来调用不用init了, 直接完成init: f=FooBar(); f.init(), 仅需 f=FooBar() 即可

```
>>> class FooBar:
...     def init(self): ←
...         self.somevar=42
...
>>> f=FooBar()
>>> f.init() ←
>>> f.somevar
42
```

修改init为__init__

```
>>> class FooBar_new:
...     def __init__(self): ←
...         self.somevar=42
...
>>> f=FooBar_new()
>>> f.somevar
42
```

重写是继承机制的一个重要方面, 对构造函数来说尤其重要; 但是与重写普通方法相比, 重写构造函数时更有可能遇到一个特别的问题: 重写构造函数时, 必须调用超类(继承的类)的构造函数, 否则可能无法正确地初始化对象

使用super函数调用超类的构造函数:

超类:

```
>>> class Bird:  
...     def __init__(self):  
...         self.hungry=True  
...     def eat(self):  
...         if self.hungry:  
...             print("Aaaaah...")  
...             self.hungry=False  
...         else:  
...             print('No, thanks!')
```

重写构造函数super:

```
>>> class SongBird(Bird):  
...     def __init__(self):  
...         super().__init__()  
...         self.sound='Squawk'  
...     def sing(self):  
...         print(self.sound)  
...
```

函数super很聪明, 因此即便有多个超类, 也只需调用函数super一次

- 基本到序列和映射协议

序列和映射基本上是元素(item)的集合, 要实现它们的基本行为(协议), 不可变对象需实现2个方法, 而可变对象需要实现4个

`__len__(self)`: 这个方法应返回集合包含的项数, 对序列来说为元素个数, 对映射来说为键-值对数, 如果`__len__`返回零, 对象在布尔上下文中将被视为假(就像空的列表, 元组, 字符串和字典一样)

`__getitem__(self, key)`: 这个方法应返回与指定键相关联的值

`__setitem__(self, key, value)`: 这个方法应以与键相关联的方式存储值, 以便以后能够使用`__getitem__`来获取, 仅当对象可变时才需要实现这个方法

`__delitem__(self, key)`: 该方法在对对象的组成部分使用`__del__`语句时被调用, 应删除与key相关联的值

对于序列, 如果键为负值, 从未尾往前数: `x[-n] == x[len(x)-n]`

若键类型不合适, 可能引发`TypeError`异常; 对于序列, 如果索引的类型正确的, 但不在允许的范围内, 应引发`IndexError`异常

使用`super().__init__(args)`继承list, 使用list所有内置函数

- 特性

```
>>> class Rectangle:  
...     def __init__(self):  
...         self.width=0  
...         self.height=0  
...     def set_size(self,size):  
...         self.width,self.height=size  
...     def get_size(self):  
...         return self.width,self.height
```

get_size和set_size是假想属性size的存取方法, 这个属性是一个width和height组成的元组; 若想使size成为真正的属性: 使用函数property

```
>>> class Rectangle:  
...     def __init__(self):  
...         self.width=0  
...         self.height=0  
...     def set_size(self,size):  
...         self.width,self.height=size  
...     def get_size(self):  
...         return self.width,self.height  
...     size = property(get_size, set_size)  
...
```

在此新版的Rectangle中, 通过调用函数property并将存取方法作为参数(获取方法在前, 设置方法在后)创建了一个特性, 然后将名称size关联到这个特性。

```
>>> r.width=10  
>>> r.height=5  
>>> r.size  
(10, 5)  
>>> r.size=150,100  
>>> r.width  
150
```

实际上, 调用函数property时, 还可以不指定参数, 指定一个参数或三或四个参数。

```
class property(object)  
    property(fget=None, fset=None, fdel=None, doc=None)  
        Property attribute.  
        |  
        fget  
            function to be used for getting an attribute value  
        fset  
            function to be used for setting an attribute value  
        fdel  
            function to be used for del'ing an attribute  
        doc  
            docstring
```

property其实并不是函数, 而是一个类. 它的实例包含一些魔法方法, 而所有的魔法都是由这些方法完成的. 这些魔法方法为 `__get__`, `__set__`, `__delete__`, 它们一道定义了所谓的描述符协议。p277

- 静态方法和类方法

略

- 迭代器

迭代(iterate)意味着重复多次, 就像循环那样. 方法 `__iter__` 返回一个迭代器, 它是包含方法 `__next__` 的对象, 而调用这个方法时可不提供任何参数. 当你调用方法 `__next__` 时, 迭代器应返回其下一个值. 如果迭代器没有可供返回的值, 应引发`StopIteration`异常. 可使用内置的便利函数 `next`, 此时, `next(it)` 与 `it.__next__()` 等效.

```
>>> class Fibs:  
...     def __init__(self):  
...         self.a=0  
...         self.b=1  
...     def __next__(self):  
...         self.a,self.b=self.b,self.a+self.b  
...         return self.a  
...     def __iter__(self):  
...         return self
```

可通过可迭代对象调用内置函数iter, 可获得一个可迭代器:

```
>>> it=iter([1,2,3])  
>>> it  
<list_iterator object at 0x10d129c50>  
>>> next(it)  
1  
>>> next(it)  
2
```

使用构造函数list显示地将迭代器转换为列表:

```
>>> class TestIterator:  
...     value=0  
...     def __next__(self):  
...         self.value+=1  
...         if self.value > 10:raise StopIteration  
...         return self.value  
...     def __iter__(self):  
...         return self  
  
>>> ti=TestIterator()  
>>> list(ti)  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- 简单生成器

```
>>> g = ((i+2) ** 2 for i in range(2,27))  
>>> g  
<generator object <genexpr> at 0x10d0e9d50>  
>>> next(g)  
16
```

其工作原理与列表推导相同, 但不是创建一个列表(即不立即执行循环), 而是返回一个生成器, 让你能够逐步执行计算. 不同与列表推导式, 这里使用的是圆括号.

```
>>> [(i+2) ** 2 for i in range(2,27)]  
[16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 48  
4, 529, 576, 625, 676, 729, 784]  
>>> sum((i+2) ** 2 for i in range(10))  
505
```

或在一对既有圆括号内(如在函数调用中)使用生成器推导, 无需再添加一对圆括号(如上).

- 递归生成器

使用两个for循环可实现两层嵌套列表, 处理任意层嵌套列表, 使用递归:

```
>>> def flatten(nested): ←  
...     try:  
...         for sublist in nested:  
...             for element in flatten(sublist):  
...                 yield element ←  
...     except TypeError:  
...         yield nested  
...
```

处理递归时存在两种情况: 基线条件和递归条件, 上式中, 基线情况下展开一个单元, 若为单个元素, 将引发TypeError异常.

```
>>> list(flatten([[1], 2], 3, 4, [5, [6, 7]], 8))  
[1, 2, 3, 4, 5, 6, 7, 8]
```

这里若nested是字符穿或类似字符串对象, 属于序列, 不会引发TypeError异常. 因此, 可在开头尝试将其与一个字符串拼接检查:

```
def flatten(nested):  
    try:  
        # 不迭代类似于字符串的对象:  
        try: nested + ''  
        except TypeError: pass  
        else: raise TypeError  
        for sublist in nested:  
            for element in flatten(sublist):  
                yield element  
    except TypeError:  
        yield nested
```

- 通用生成器

生成器是包含了关键字yield的函数, 但被调用时不会执行函数体内的代码, 而是返回一个迭代器, 每次请求值, 都将执行生成器的代码, 直到遇到yield或return. **yield**意味着应生成一个值, 而**return**意味着生成器停止执行(即不再生成值, 仅当在生成器调用**return**时, 才能不提供任何参数).

生成器由两个单独部分组成: 生成器的函数和生成器的迭代器. 生成器的函数由def语句定义, 其中包含yield. 生成器的迭代器就是这个函数返回的结果.

外部世界可访问生成器的方法send, 类似于next, 但接受一个参数.

注意: 如果要以某种形式返回值, 就不管三七二十一, 将其用圆括号括起来.

方法throw: 用于在生成器中(yield表达式处)引发异常, 调用时提供一个异常类型, 一个可选值和一个 traceback对象

方法close: 用于停止生成器, 调用时无需提供任何参数.

- 八皇后问题

状态表示: 使用元组(或列表)来表示可能的解(或其一部分), 其中每个元素表示相应中皇后的所在位置(即列). 例如: state[0]==3, 表示第1行的皇后放置在第4列(从0计算).

检测冲突: 函数conflict接受既有的皇后位置, 并确定下一个皇后的位置是否冲突:

总结:

- 构造函数: 很多面向对象语言中都有构造函数, 对于你自己编写的每个类, 都可能需要为它实现一个构建函数. 构建函数名为 `_init_`, 在对象构建后被自动调用.
- 迭代器: 简单地说, 迭代器是包含方法 `_next_` 的对象, 可用于迭代一组值. 没有更多的值可供迭代时, 方法 `_iter_`, 它返回一个像序列一样可用于for循环中的迭代器. 通常, 迭代器也是可迭代的, 即包含返回迭代器本身的方法 `_iter_`.
- 生成器: 生成器的函数是包含关键字 `yield` 的函数, 它在被调用时返回一个生成器, 即一种特殊的迭代器. 要与活动的生成器交互, 可使用方法 `send`, `throw` 和 `close`.

`iter(obj)`: 从可迭代对象创建一个迭代器

`next(it)`: 让迭代器前进一步并返回下一个元素

`property(fget, fset, fdel, doc)`: 返回一个特性; 所有参数都是可选的

`super(class, obj)`: 返回一个超类的关联实例

- 模块就是程序

创建简单模块hello.py: `print("Hello, world!")`

```
import sys; sys.path.append("/dir/to/hello.py")  
  
import hello: Hello, world!
```

发现模块hello.py所在目录会出现一个名为 `_pycache_` 的子目录, 包含处理后的文件, Python能够更高效地处理它们. 重复导入不会执行代码, 因为模块不是用来执行操作的, 而是定义变量, 函数, 类等. 鉴于定义只需做一次, 因此导入模块多次和导入一次的效果相同.

若一定要重新加载模块, 可使用模块importlib中的函数reload, 接受一个参数(需要重新加载的模块), 并返回重新加载的模块:

```
import importlib  
  
hello=importlib.reload(hello): Hello, world!
```

- 模块使用来下定义的

在模块中定义函数, 像处理模块那样, 让程序可用后, 可使用 `python -m programe args`, 使用命令行参数args来执行程序programe

```
# hello2.py  
def hello():  
    print("Hello, world!")
```

```
>>> import hello2  
>>> hello2.hello()  
Hello, world!
```

```
carlos@huizhendeMacBook-Pro 21:30:4  
/Scripts_practice/python_scripts  
$
```

可在模块中加入测试代码, 例如在 `def hello():...` 后加上 `hello()`, 那么在 `import` 时, 若正确就会执行 `hello()`. 但这不是一个好的结果, 若要避免, 关键是检查模块是否作为程序运行还是导入另一个程序. 为此, 使用变量 `__name__`

```
>>> __name__  
'__main__'  
>>> hello2.__name__  
'hello2'  
>>> █
```

在主程序(包括解释器的交互式提示符), 变量 `__name__` 的值为 `__main__`; 而在导入的模块中, 该值被设置为该模块的名称.

```
腠 hello4.py > ...  
1 def hello():  
2     print("Hello, world!")  
3  
4 def test():  
5     hello()  
6  
7 if __name__ == '__main__':test()
```

```
>>> import hello4  
>>> hello4.hello()  
Hello, world!  
>>> hello4.test()  
Hello, world!  
>>> █
```

```
carlos@huizhendeMacBook-Pro 21:43:1  
/Scripts_practice/python_scripts  
$py3 hello4.py  
Hello, world!  
carlos@huizhendeMacBook-Pro 21:46:1  
/Scripts_practice/python_scripts
```

如果将该模块当程序运行, 将执行函数 `hello()`, 若导入它, 则像其他普通模块一样; 同时使用了 `if` 函数, 可通过 `hello4.test()` 独立测试.

- 让模块可用

根据模块目录 `sys.path` 将模块放在正确的位置:

```
>>> import sys, pprint  
>>> pprint.pprint(sys.path)  
['',  
 '/usr/local/Cellar/python/3.7.5/Frameworks/Python.framework/Versions/3.7/lib/python37.zip',  
 '/usr/local/Cellar/python/3.7.5/Frameworks/Python.framework/Versions/3.7/lib/python3.7',  
 '/usr/local/Cellar/python/3.7.5/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload',  
 '/Users/carlos/Library/Python/3.7/lib/python/site-packages',  
 '/usr/local/lib/python3.7/site-packages']
```

告诉解释器到哪里去查找

```
export PYTHONPATH=$PYTHONPATH:~/python
```

- 包

为组织模块, 可将其编组为包(package). 包其实就是另一种模块, 但有趣的是他们可包含其他模块. 模块存储在扩展名为 `.py` 的文件中, 而包则是一个目录. 要被 `python` 视为包, 目录必须包含文件 `__init__.py`. 如果像普通模块一样导入包, 文件 `__init__.py` 的内容就将是包的内容. 例如

```
>>> import constants  
>>> print(constants.PI)  
3.14  
>>> █
```

```
1 PI=3.14  
2  
s/__init__.py lines 1-2/2 (END)█
```

那么, 若要将模块加入包中, 只需将模块文件放在包的目录中即可:

```
>>> import constants.hello2  
Hello, world!  
>>> from constants import hello2  
>>> 
```

```
carlos@huizhendeMacBook-Pro 22:07  
:51 /Scripts_practice/python_scripts  
$ls constants/  
__init__.py      hello2.py  
__pycache__  
carlos@huizhende MacBook Pro 22:07
```

仅导入 `constants` 时, 仅能使用 `__init__.py` 中的值, 不能使用模块 `hello2.py`, 需单独导入.

- 模块包含什么

要查明模块包含哪些东西, 可使用函数 `dir`, 它列出对象的所属性(对于模块, 它列出所有的函数, 类, 变量等). 在这些名称中, 有几个以下划线打头, 根据约定, 这意味着它们并非供外部使用.

```
>>> [n for n in dir(copy) if not n.startswith("__")]
['Error', 'copy', 'deepcopy', 'dispatch_table', 'error']
>>> dir(copy)
['Error', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', '_copy_dispatch', '_copy_immutable', '_deepcopy_atomic', '_deepcopy_d
ict', '_deepcopy_dispatch', '_deepcopy_list', '_deepcopy_method', '_deepcopy_tuple', '_keep_alive'
, '_reconstruct', 'copy', 'deepcopy', 'dispatch_table', 'error']
```

`dir(copy)` 返回清单中包含 `__all__`, 为模块内部设置, 避免模块包含大量其他程序不需要变量, 函数和类, 若不设置 `__all__`, 则会在 `from copy import *` 导入所有不以下划线打头的全局名称.

- 使用 `help` 获得帮助

```
help(copy.copy) / help(copy)
```

`__doc__` 模块文档字符串就是函数开头编写的字符串, 用于对函数进行说明, 而函数的属性 `__doc__` 可能包含这个字符串, 同时, 模块也可能有文档字符串(位于模块的开头), 而类也可能如此(位于类的开头).

```
def store_person(db):
    """
    让用户输入数据并将其存储到shelf对象中
    """
```

```
>>> print(copy.__doc__)
Shallow copy operation on arbitrary Python objects.

See the module's __doc__ string for more info.
```

`deepcopy(x)` 创建 `x` 的属性的副本并依次类推; 而 `copy(x)` 只复制 `x`, 并将副本的属性关联到 `x` 的属性值

- 使用源代码

但要真正理解 python 语言, 可能需要了解一些不阅读源代码就无法了解的事情, 事实上, 要学习 python, 阅读源代码时除手动编写代码外的最佳方式

```
>>> print(copy.__file__)
/usr/local/Cellar/python/3.7.5/Frameworks/Python.framework/Versions/3.7/lib/python3.7/copy.py
>>> 
```

- 标准库

模块 `sys`

`argv`: 命令行参数, 包括脚本名

`exit([arg])`: 退出当前程序, 可通过可选参数指定返回值或错误消息

`modules`: 一个字典, 将模块名映射到加载的模块

`path`: 一个列表, 包含要在其中查找模块的目录的名称

`stdin`: 标准输入流, 一个类似于文件的对象

`stdout`: 标准输出流, 一个类似于文件的对象

`stderr`: 标准错误流, 一个类似于文件的对象

模块os, 可用于访问多个操作系统服务, 除此之外, os及其子模块os.path还包含多个查看, 创建和删除目录及文件的函数, 以及一些操作路径的函数(例如, `os.path.split/os.path.join`让你在大多数情况下都可以忽略`os.pathsep`)

`environ`: 包含环境变量的映射

`system(command)`: 在子shell中执行操作系统命令

`sep`: 路径中使用的分隔符

`pathsep`: 分隔不同路径的分隔符

`linesep`: 行分隔符('`\\n`', '`\\r`'或`'\\r\\n`')

模块time

元组(2008,1,21,12,2,56,0,21,0)表示2008年1月21日12时2分56秒, 星期一, 2008年的第21天(不考虑夏令时)

python日期元祖中的字段(索引从0开始): 年, 月, 日, 时, 分, 秒(0-61), 星期(0-6), 儒略日(1-366), 夏令时(0/1/-1)

模块time中的重要函数

`asctime([tuple])`: 将时间元组转换为字符串

`localtime([secs])`: 将秒数转换表示为当地时间的日期元组

`mktme(tuple)`: 将时间元组转换为当地时间

`sleep(secs)`: 休眠(什么都不做)secs秒

`strptime(string[, format])`: 将字符串转换为时间元组

`time()`: 当前时间(从新纪元开始后的秒数, 以UTC为准)

```
>>> time()
1581917463.738392
>>> localtime()
time.struct_time(tm_year=2020, tm_mon=2, tm_mday=17, tm_hour=13, tm_min=31,
tm_sec=7, tm_wday=0, tm_yday=48, tm_isdst=0)
>>> asctime(localtime())
'Mon Feb 17 13:31:26 2020' ←
>>> mktme(localtime())
1581917497.0
```

模块random包含生成伪随机数的函数(其背后的系统是可预测的), 有助于编写模拟程序或生成随机输出的程序. 若要求真正的随机, 考虑使用模块os中的函数urandom.

`random()`: 返回一个0-1(含)的随机实数

```
getrandbits(n) : 以长整数方式返回n个随机的二进制位
```

```
uniform(a,b) : 返回一个a-b(含)的随机实数
```

```
randrange([start],stop,[step]) : 从range(start,stop,step)中随机地选择一个数
```

```
choice(seq) : 从序列seq中随机地选择一个元素
```

```
shuffle(seq[, random]) : 就地打乱序列seq
```

```
sample(seq,n) : 从序列seq中随机地选择n个值不同的元素
```

```
>>> date1  
(2016, 1, 1, 0, 0, 0, -1, -1, -1)  
>>> date2  
(2017, 1, 1, 0, 0, 0, -1, -1, -1)  
>>> time1=mktime(date1)  
>>> time2=mktime(date2)  
>>> random_time=uniform(time1,time2)  
>>> asctime(localtime(random_time))  
'Fri Feb 12 11:34:54 2016'  
---
```

```
>>> values = list(range(1, 11)) + 'Jack Queen King'.split()  
>>> suits = 'diamonds clubs hearts spades'.split()  
>>> deck = ['{} of {}'.format(v, s) for v in values for s in suits]
```

```
shuffle(deck); while deck:input(deck.pop())
```

- 集合, 堆和双端队列

以前集合是由模块sets中的Set类实现的, 较新的版本中, 集合是由内置类set实现, 可直接创建集合, 无需导入模块sets

```
>>> set(range(10))  
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
>>> set()  
set()  
>>> {0,1,2,3,4,5,5,6,7,8,0}  
{0, 1, 2, 3, 4, 5, 6, 7, 8}
```

集合主要用于成员资格检查, 因此将忽略重复的元素, 与字典一样, 集合中元素的排列顺序是不确定的. 集合可用于各种标准集合操作, 如并集和交集, 为此可使用对整数执行按位操作的运算符(参考附录B)

```
>>> a = { 1, 2,3}  
>>> b = {2,3,4}  
>>> a.union(b)  
{1, 2, 3, 4}  
>>> a|b  
{1, 2, 3, 4}
```

计算两个集合的并集的函数时, 可使用set中方法union的未关联版本, 搭配reduce使用(接受的参数和map类似, 一个函数f, 一个list, 但是行为和map不同, reduce传入的函数f必须接受两个参数, reduce对list的每个元素反复调用函数f, 并返回最终结果值; map每次调用函数后, 返回包含返回值的列表, py2为列表, py3为迭代器):

```
>>> import functools
>>> my_sets=[]
>>> for i in range(10):
...     my_sets.append(set(range(i,i+5)))
...
>>> functools.reduce(set.union,my_sets)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}
```

```
>>> for i in map(set.union,my_sets):
...     print(i)
...
{0, 1, 2, 3, 4}
{1, 2, 3, 4, 5}
{2, 3, 4, 5, 6}
{3, 4, 5, 6, 7}
{4, 5, 6, 7, 8}
{5, 6, 7, 8, 9}
{6, 7, 8, 9, 10}
{7, 8, 9, 10, 11}
{8, 9, 10, 11, 12}
{9, 10, 11, 12, 13}
>>> █
```

集合时可变的, 因此不能用作字典中的键. 另一个问题, 集合只能包含不可变的值, 因此不能包含其他集合. 可使用frozenset类型, 它表示不可变的集合:

```
>>> a=set()
>>> d=set([1,2,3])
>>> d
{1, 2, 3}
>>> a.add(frozenset(d))
>>> a
{frozenset({1, 2, 3})}
```

堆(heap), 一种优先队列. 优先队列让你能够以任意顺序添加对象, 并随时(可能是在两次添加对象之间)找出(并删除)最小的元素. 相比于列表方法min, 这样做的效率要高得多. 堆操作函数模块 `heapq`, 包含6个函数:

`heappush(heap,x)`: 将x压入堆中

`heappop(heap)`: 从堆中弹出最小的元素

`heapify(heap)`: 让列表具备堆特征

`heappreplace(heap,x)`: 弹出最小的元素, 并将x压入堆中

`nlargest(n,iter)`: 返回iter中n个最大的元素

`nsmallest(n,iter)`: 返回iter中n个最小的元素

堆特征: 位于i处的元素总是大于位置 `i//2` 处的元素(反过来说就是小于位置 `i*2` 和 `i*2 + 1` 处的元素), 这是底层堆算法的基础, 称为堆特征(**heap property**).

在需要按添加元素的顺序进行删除时, 双端队列很有用. 在模块collections中, 包含类型deque以及其他几个集合(collection)类型. 和集合(set)一样, 双端队列也是从可迭代对象创建的:

```

>>> from collections import deque
>>> q=deque(range(5))
>>> q
deque([0, 1, 2, 3, 4])
>>> q.append(5)
>>> q.appendleft(6)
>>> q
deque([6, 0, 1, 2, 3, 4, 5])
>>> q.pop()
5
>>> q.popleft()
6
>>> q.rotate(3)
>>> q
deque([2, 3, 4, 0, 1])
>>> q.rotate(-1)
>>> q
deque([3, 4, 0, 1, 2])

```

tmp_back_R.py

var.py

OUTLINE

The active editor
cannot provide
outline information.

双端队列支持队首(左端)高效弹出和附加元素, 还可以高效地旋转元素(将元素向右或向左移, 并在到达一端时环绕到另一端); 此外还支持 `extend` 和 `extendleft`, 注意用于 `extendleft` 的可迭代对象中的元素将按照相反的顺序出现在双端队列中.

```

>>> q.extendleft(q)
>>> q
deque([2, 1, 0, 4, 3, 2, 1, 0, 4, 3, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2])
>>>

```

- shelve和json

对于模块shelve, 唯一感兴趣的函数 `open`. 这个函数将一个文件名作为参数, 并返回一个 `Shelf` 对象, 供你用来存储数据. 可像操作普通字典那样操作(只是键必须为字符串), 操作完毕(并将所做的修改存盘)时, 可调用其他方法 `close`.

```

>>> import shelve
>>> s = shelve.open('test.dat')
>>> s['x'] = ['a', 'b', 'c']
>>> s['x'].append('d') ←
>>> s['x'] ←
['a', 'b', 'c']

```

这里列表['a','b','c']被存储到了s的'x'键下; 将'd'附加到这个新列表末尾, 但这个修改后的版本未被存储; 最后再次获得原来版本, 其中没有'd'.

要正确地修改使用模块shelve存储的对象, 必须将获取的副本赋给一个临时变量, 并在修改这个副本后再次存储:

```

>>> temp = s['x'] ←
>>> temp.append('d') ←
>>> s['x'] = temp ←
>>> s['x']
['a', 'b', 'c', 'd']

```

另一个方法是, 将函数open的参数`writeback`设置为True, 使得shelf对象读取或赋给它的所有数据结构都将保存到内存(缓存)中, 并等到你关闭shelf对象(`temp.close()`)时才将它们写入磁盘

(`t=open('test.dat', writeback=True)`).

- re

正则表达式是可匹配文本片段的模式. 最简单的正则表达式为普通字符串, 与它自己相匹配.

通配符: 点好. 除了换行符外和所有其他字符匹配, 仅匹配一个字符, 而不与零或两个字符匹配

转义: 就是让特殊字符的行为与普通字符一样, `python\\org` 只与 `python.org` 匹配. 为表示模块re要求的单个反斜杠, 需要在字符串中书写两个反斜杠, 让解释器对其进行转义(包含两层转义: 解释器执行的转义和模块re执行的转义). 也可以使用原始字符串 `r'python\\org'`

字符集: `[a-zA-Z0-9]` 表示匹配大写字母, 小写字母和数字, 也只能匹配一个字符; 若要排除可在开头加上`^`: `[^abc]` 表示匹配除a/b/c以外的其他任何字符; 同样, 对于右括号`()`和连字符`-`, 要么放在字符集开头, 要么使用反斜杠对其进行转义, 连字符也可放在末尾.

二选一和子模式: 使用圆括号限定选择部分, 使用管道字符`(|)`隔开子模式, `p(ython|erl)`

可选模式和重复模式: 通常在子模式后面加上问号, 可将其指定为可选的:

`(pattern)?`: 一次或零次

`(pattern)*`: 重复零次或1次或多次

`(pattern)+`: 重复1或多次

`(pattern){m,n}`: 重复m~n次

字符串的开头和末尾: 脱字符`(^)`指定开头匹配; 指定字符串末尾匹配使用美元符号`(\$)`

`compile(pattern[, flags])`: 根据包含正则表达式的字符串创建模式对象, 提高匹配效率

`search(pattern, string[, flags])`: 在字符串中查找模式

`match(pattern, string[, flags])`: 在字符串开头匹配模式, 仅匹配开头, 若完全匹配后加\$符号

`split(pattern, string[, maxsplit=0])`: 根据模式来分隔字符串

`findall(pattern, string)`: 返回一个列表, 其中包含字符串中所有与模式匹配的子串

`sub(pat, repl, string[, count=0])`: 将字符串中与模式pat匹配的子串都替换成repl

`escape(string)`: 对字符串中所有的正则表达特殊字符都进行转义

```
>>> pattern=compile("[a-z]{2}")
>>> pattern.findall("a b c d dd sd r")
['dd', 'sd']

>>> some_text='alpha, beta,,,gama delat'
>>> split('[, ]+',some_text)
['alpha', 'beta', 'gama', 'delat']
```

如果模式包含圆括号, 将在分割得到的子串之间插入括号中的内容

```
>>> split('o(o)', 'foobar')
['f', 'o', 'bar']
```

函数re.sub从左往右将与模式匹配的子串替换为指定内容:

```
>>> pat='{name}'
>>> text='Dear {name}...'
>>> sub(pat, 'Mr. Gumby',text)
'Dear Mr. Gumby...'
```

re.escape是一个工具函数, 对于字符串中所有可能被视为正则表达式运算符的字符进行转义:

```
>>> re.escape('www.python.org')    >>> re.escape("www.python.org")
'www\\\.python\\\.org'           'www\\\.python\\\.org'
>>> re.escape('But where is the ambiguity?')
'But\\\" where\\\" is\\\" the\\\" ambiguity\\\"'
```

匹配对象和编组: 查找与模式匹配的子串的函数都在找到时返回MatchObject对象. 这种对象包含与模式匹配的子串的信息, 还包括模式的哪部分与子串的哪部分匹配的信息, 这些子串部分称为编组(group). 编组就是放在圆括号内的子模式, 它们时根据左边的括号数编号的, 其中编组0指的是整个模式.

```
'There (was a (wee) (cooper)) who (lived in Fyfe)'
```

包含如下编组:

```
0 There was a wee cooper who lived in Fyfe
1 was a wee cooper
2 wee
3 cooper
4 lived in Fyfe
```

re匹配对象的重要方法:

`group([group1,...])`: 获得与给定子模式(编组)匹配的子串

`start([group])`: 返回与给定编组匹配的子串的起始位置

`end([group])`: 返回与给定编组匹配的子串的终止位置(与切片一样, 不包含终止位置)

`span([group])`: 返回与给定编组匹配的子串的起始和终止位置

```
>>> m=match(r'www\.(.*)...{3}','www.python.org')
>>> m.group(1)
'python'
>>> m.start(1)
4
>>> m.end(1)          ←
10
>>> m.span(1)
(4, 10)
```

替换中的组号和函数: 若只是字符串替换, 也可使用字符串方法 `replace(old, new[, count])`; 为利用 `sub` 函数的强大功能, 最简单的方法是在替换字符串中使用组号

要让正则表达式容易理解, 一种方法是在调用模块 `re` 中的函数时使用标志 `VERBOSE`. 这使得能在模式中添加空白(空格, 制表符, 换行符等), 而 `re` 将忽略它们, 除非将它放在字符类中或使用反斜杠对其进行转义, 在这样的正则表达式中, 你还可以添加注释:

```
>>> emphasis_pattern = re.compile(r'''
... \*                  # 起始突出标志——一个星号
... (
... [^*\n]+              # 与要突出的内容匹配的编组的起始位置
... )                  # 编组到此结束
... \*                  # 结束突出标志
...     ''', re.VERBOSE)
```

模版(template)是一种文件, 可在其中插入具体的值来得到最终的文本. Python 提供了一种高级模版机制: 字符串格式设置, 使用正则表达式可让这个系统更加高级.

[x = 2] ←
[y = 3] ←
The sum of [x] and [y] is [x + y]. →

```
>>> "The sum of 7 and 9 is {}".format(7+9,4+3)  
'The sum of 7 and 9 is 7'  
>>> "The sum of 7 and 9 is {}".format(7+9,4+3)  
'The sum of 7 and 9 is 16'
```

- 其他有趣标准模块

`argparse`: 提供功能齐备的命令行界面

`cmd`: 编写类似Python交互式解释器的命令行解释器

`csv`: csv指的是逗号分隔的值(comma-separated values), 模块`csv`帮助轻松读写csv文件, 以非常透明的方式处理CSV格式的一些棘手部分

`datetime`: 支持特使的日期和时间对象, 并让你能够以各种方式创建和合并这些对象

`difflib`: 确定两个序列的相似程度, 帮组从很多序列中找出与指定序列最为相似的序列. 可使用`difflib`来创建简单的搜索程序

`enum`: 枚举类型是一种只有少数几个可能取值的类型

`hashlib`: 使用该模块可计算字符串的小型“签名”(数), 可用来计算大型文件的签名, 这个模块在加密和安全领域有很多用途

`functools`: 在调用函数时只提供部分参数(部分求值, partial evaluation), 以后再填充其他的参数. 在Python3.0中, 这个模块包含`filter`和`reduce`

`itertools`: 包含大量用于创建和合并迭代器(或其他可迭代对象)的工具, 其中包括可以串接可迭代对象, 创建返回无限连续整数的迭代器(类似于`range`, 但没有上限), 反复遍历可迭代对象以及具有其他作用的函数

`logging`: 提供了一系列标准工具, 可用于管理一个或多个中央日志, 它还支持多种优先级不同的日志消息

`statistics`: 使用该模块进行高级运算

`timeit, profile, trace`: 模块`timeit`(和配套的命令脚本)是一个测量代码执行时间的工具; 模块`profile`(和配套模块`pstats`)可用于对代码段的效率进行更全面的分析; 模块`trace`可帮助你进行覆盖率分析(即代码的哪些部分执行了, 哪些部分没有执行), 在编写测试代码时很有用

- 打开文件

调用模考`io`中的`open`函数, 通过指定模式来显示指出这点: `r` 读取模式(默认值); `w` 写入模式; `x` 独占写入模式; `a` 附加模式; `b` 二进制模式(与其他模式结合使用); `t` 文本模式(默认值, 与其他模式结合使用); `+` 读写模式(与其他模式结合使用)

`w` 写入模式能够在文件不存在时创建它, 此时打开的文件, 既有内容将被删除(截断), 并从文件开头处写入; 若要在既有文件末尾继续写入, 可使用附加模式 `x` 独占写入模式会在文件已经存在时引发`FileExistsError`异常; `+` 可与其他任何模式结合起来使用, 表示既可读取也可写入, 例如`r+`表示打开进行读写, `w+`则会截断文本

```
>>> f = open("test","a")
>>> f
<_io.TextIOWrapper name='test' mode='a' encoding='UTF-8'>
```

- 读取和写入

write:

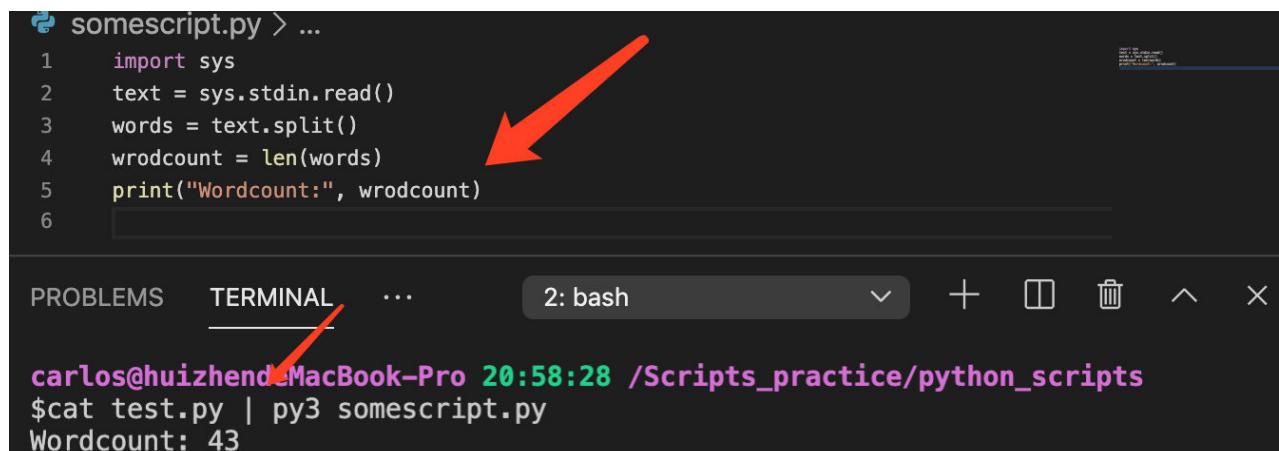
```
>>> f = open('somefile.txt', 'w')
>>> f.write('Hello, ')
7
>>> f.write('World!')
6
>>> f.close()
```

read:

```
>>> f = open('somefile.txt', 'r')
>>> f.read(4)
'Hell'
>>> f.read()
'o, World!'
```

- 使用管道重定向输出

例如编写脚本统计单词数目:



```
somescript.py > ...
1 import sys
2 text = sys.stdin.read()
3 words = text.split()
4 wrodcoumt = len(words)
5 print("Wordcount:", wrodcoumt)
```

PROBLEMS TERMINAL ... 2: bash + ×

```
carlos@huizhende:MacBook-Pro 20:58:28 /Scripts_practice/python_scripts
$cat test.py | py3 somescript.py
Wordcount: 43
```

我们将文件都视为流, 只能按顺序从头到尾读取. 实际上, 可在文件中移动, 只访问感兴趣的部分(称为随机存取). 使用文件对象两个方法: seek 和 tell

seek 设置偏移量:

```
>>> f = open(r'C:\text\somefile.txt', 'w') 路径
>>> f.write('01234567890123456789')
20
>>> f.seek(5) ←
5
>>> f.write('Hello, World!')
13
>>> f.close()
>>> f = open(r'C:\text\somefile.txt')
>>> f.read()
'01234Hello, World!89'
```

`tell` 返回当前位于文件的什么位置:

```
>>> f = open(r'C:\text\somefile.txt')
>>> f.read(3)
'012'
>>> f.read(2)
'34'
>>> f.tell() ←
5
```

- 读取和写入行

要读取一行(从当前位置到下一个分行符的文本), 使用方法 `readline`, 调用时不提供任何参数(读取一行并返回它); 也可提供一个非负整数, 指定 `readline` 最多可读取多少个字符. 若要读取文本中所有行, 并以列表方式返回, 可使用方法 `readlines`; 方法 `writelines` 与 `readlines` 相反, 接受一个字符串列表(可以是任何序列或可迭代对象), 并将这些字符串都写入到文件(或流)中.

- 关闭文件

对于写入过的文件, 一定要将其关闭, 因为 Python 可能缓冲你写入的数据(将数据暂时存储在某个地方, 以提供效率). 因此如果程序因某种原因崩溃, 数据可能根本不会写入到文本中. 安全的做法是, 使用完文件后就将其关闭. 如果要重置缓冲, 让所做的修改反应到磁盘文件中, 但又不想关闭文件, 可使用方法 `flush`.

```
# 在这里打开文件
try:
    # 将数据写入到文件中
finally:
    file.close()
```

专门为设计的语句, `with` 语句. 能够打开文件并将其赋给一个变量. 在语句体中, 你将数据写入文件(可能是做其他事情). 到达该语句末尾时, 将自动关闭文件, 即便出现异常亦如此

```
with open("somefile.txt") as somefile:
    do_something(somefile)
```

- 使用文件的基本方法

```
open("test.py", "r").read()
open("test.py", "r").readline()
open("test.py", "r").readlines()
open("t.txt", "a").write("This \n is \n no \n haiku")
```

```
>>> f = open(r'C:\text\somefile.txt')
>>> lines = f.readlines()
>>> f.close()
>>> lines[1] = "isn't a\n"
>>> f = open(r'C:\text\somefile.txt', 'w')
>>> f.writelines(lines) ←
>>> f.close()
```

- 迭代文件内容

一种最常见的文本操作是迭代其内容，并在迭代过程中反复采用某种措施

```
def process(string):
    print('Processing:', string)
```

更有用的实现包括将数据存储在数据结构中，计算总和，使用模块re进行模式替换以及添加行号

- 每次一个字符(或字节)

最简单(也可能是最不常见)的文件内容迭代方式是，在while循环中使用方法read

```
1  with open("t.txt") as f:
2      char=f.read(1)
3      while char: ←
4          print(char) ## some processes
5          char=f.read(1) ←
```

或使用while True/break技巧

```
with open("t.txt") as f:
    while True: ←
        char=f.read(1)
        if not char: break ←
        print([char])
```

- 每次一行

```
with open(filename) as f:
    while True:
        line = f.readline() ← readline,仅仅读首行
        if not line: break
        process(line)
```

- 读取所有内容

如果文件不太大，可一次读取整个文件；为此，可使用方法read并不提供任何参数(将整个文件读到一个字符串中)

```
with open(filename) as f:
    for char in f.read():
        process(char)
```

也可使用方法readlines(将文件读取到一个字符串列表中，其中每个字符串都是一行)

```
with open(filename) as f:
    for line in f.readlines():
        process(line)
```

- 使用fileinput实现延迟行迭代

Python中，在可能的情况下，应首选for循环；模块fileinput会负责打开文件，只需要提供一个文件名即可

```
>>> for line in fileinput.input("t.txt"):
...     print(line,end="")
...
This
is
no
haikunumber = 35
guess=int(input("Enter your guess number: "))
if guess == number:
```

- 文件迭代器

文件实际上是可以迭代的, 这意味着可在for循环中直接使用它们来迭代

```
>>> with open("t.txt") as f:  
...     for line in f:  
...         print(line)  
...  
This  
  
is  
  
no
```

在不将文件对象赋给变量的情况下迭代文件

```
>>> for line in open("t.txt"):  
...     print(line,end="")  
...  
This  
is  
no  
haikunumber = 35
```

和其他文件一样, `sys.stdin` 也是可以迭代的, 因此要迭代标准输入中的所有行

```
>>> import sys  
>>> for line in sys.stdin:  
...     print(line)  
...  
stdin lines 1  
stdin lines 1
```

另外可对迭代器做的事情基本上都可以对文件做, 如(使用`list(open(filename))`)将其转换为字符列表, 其效果与使用`readlines`相同

```
f = open('somefile.txt','w')  
>>> print('First', 'line', file=f)  
>>> print('Second', 'line', file=f)  
>>> print('Third', 'and final', 'line', file=f)  
>>> f.close()  
>>> lines = list(open('somefile.txt'))  
>>> lines  
['First line\n', 'Second line\n', 'Third and final line\n']  
>>> first, second, third = open('somefile.txt')  
>>> first  
'First line\n'  
>>> second  
'Second line\n'  
>>> third  
'Third and final line\n'
```

注意, 使用`print`来写入文件, 这将自动在提供的字符串后面添加换行符; 写入文件后将其关闭, 以确保数据得以写入磁盘.

Miscellaneous

- Jupyter Notebook

Jupyter Notebook是一个JSON文件, 拥有概况, 同时包含一系列输入和输出的有序cells, 这些cells包含了代码, MarkDown文件, 数学表达式, 图形, 表格和媒体. Jupyter Notebooks使用 .ipynb 文件后缀.

Jupyter Notebook可以转换成一系列不同的格式, 例如, HTML, slides, PDF, Markdown甚者Python. 该文件非常方便构建逐步的交互Python程序. 非常适合于数据分析和绘图.

```
Eduardo.Freitas@tis-de-55-w10 C:\Users\EduardoFreitas
$ pip install jupyter []
```

```
cmd.exe*[64]:45304
```

```
« 190526[64] 1/1 [+]
```

```
NUM PRInt
```

```
80x25 (23,24)
```

```
25V
```

```
27824
```

```
100%
```

安装并加载

```
Eduardo.Freitas@tis-de-55-w10 C:\Users\EduardoFreitas
$ jupyter notebook
```

```
print (2+3); Shift+Enter, return 5, 交互式形式
```

- % (String Formatting Operator)

根据指定的格式来格式化字符串: `%[key][flags][width][.precision][length type] conversion`
`type % values`

`%` 是必须的, 标志着指定的开始; `key` 可选, 由括号包围的字符; `flags` 可选, 转换字符; `width` 可选, 最小的字符宽度, 若指定为 `*` 表示实际宽度来自元组中的下一个值宽度; `precision` 可选, 以点开头随后数字表示精度 `.3`; `length type` 可选, 长度修订符号; `conversion type`: 可选, 转换类型; `values` 必须, 数字, 字符串或包含值的一个变量, 用于替代转换类型.

`conversion flags`: `#` 表示使用替换模式(alternate form); `0` 针对数字做0填充宽度; `-` 左对称; 空格, 针对正数左边使用空格填充; `+` 代替空格填充.

`conversion types`: `d` 含符号整数; `i` 含符号整数; `e/E` 浮点指数格式; `f/F` 浮点十进制格式; `g/G` 浮点格式; `r` 字符串, 使用 `repr()` 转换字符; `s` 字符串, 使用 `str()` 转换字符; `%` 不转换, 结果输出 `%` 字符.

```
print '(%(language)s has %(number)03d quote types.' % {"language": "Python",
"number": 2})
```

```
Python has 002 quote types.
```

```
print ("%s %s", ("foo", "bar"))
```

```
foo bar
```

```
dct = {"foo": 10, "bar": 20}
```

```
print("%(foo)s" % dct)
```

```
10
```

```
print ("%s" % "aaa")
```

```
aaa
```

