



# Python 十大基础专题.pdf

作者：zhenguo

版权归属：zhenguo(Python与算法社区 作者)

此pdf的创作细节会同步在下面公众号，欢迎扫码关注：



以下文章全部首发在我的公众号，文章链接如下：

1[我的施工计划](#)

2[数字专题](#)

3[字符串专题](#)

4[列表专题](#)

5[流程控制专题](#)

6[编程风格专题](#)

7[函数使用](#)

8[面向对象编程\(上篇\)](#)

9[面向对象编程\(下篇\)](#)

## 10 大数据结构

现整理汇总到一起以方便读者学习。

# Python语言简介

作为开篇，简要总结下 Python 语言：

Python语言 1989 年由 Guido van Rossum 编写，一名荷兰籍程序员：



Python可以应用在众多的 领域 中：

数据分析、组件集成、网络服务、图像处理、数值计算和科学计算等领域。

Python应用的 知名公司 有：

Youtube、Dropbox、BT、知乎、豆瓣、谷歌、百度、腾讯、汽车之家等。

Python可以做的 `工作` 有：

自动化运维、测试、机器学习、深度学习、数据分析、爬虫、Web等

通常使用最广泛的是 `CPython` 编译器，它将源文件（`py`文件）转换成字节码文件（`pyc`文件），然后运行在Python虚拟机上。

## 一、Python数字专题

- 整数
  - Python2 有取值范围，溢出后自动转为长整型
  - **Python3** 中为长整型，无位数限制 理论上内存有多大，位数可能就有多大
- 长整数
  - Python2 中单独对应 Long 类型
  - Python3 中不再有Long，直接对应 int
- 浮点数
  - 带小数的数字
  - 如果不带数字，可能有 e 和 E
  - 复数
  - 高数中复数
  - 结构为：`1+2j`

下面是常用的数字相关的操作：

### 1 / 返回浮点数

即便两个整数，`/` 操作也会返回浮点数

```
In [1]: 8/5  
Out[1]: 1.6
```

### 2 // 得到整数部分

使用 `//` 快速得到两数相除的整数部分，并且返回整型，此操作符容易忽略，但确实很实用。

```
In [2]: 8//5
Out[2]: 1

In [3]: a = 8//5
In [4]: type(a)
Out[4]: int
```

## 3 % 得到余数

% 得到两数相除的余数：

```
In [6]: 8%5
Out[6]: 3
```

## 4 \*\* 计算乘方

\*\* 计算几次方

```
In [7]: 2**3
Out[7]: 8
```

## 5 交互模式下的\_

在交互模式下，上一次打印出来的表达式被赋值给变量 `_`

```
In [8]: 2*3.02+1
Out[8]: 7.04

In [9]: 1+_
Out[9]: 8.04
```

## 6 十转二

将十进制转换为二进制：

```
>>> bin(10)
'0b1010'
```

## 7 十转八

十进制转换为八进制：

```
>>> oct(9)
'0o11'
```

## 8 十转十六

十进制转换为十六进制：

```
>>> hex(15)
'0xf'
```

## 9 转为浮点类型

整数或数值型字符串转换为浮点数

```
>>> float(3)
3.0
```

如果不能转化为浮点数，则会报 `ValueError`：

```
>>> float('a')
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    float('a')
ValueError: could not convert string to float: 'a'
```

## 10 转为整型

`int(x, base=10)`

`x` 可能为字符串或数值，将 `x` 转换为整数：

```
>>> int('12', 16)
18
```

## 11 商和余数

分别取商和余数

```
>>> divmod(10, 3)
(3, 1)
```

## 12 幂和余同时做

pow 三个参数都给出，表示先幂运算再取余：

```
>>> pow(3, 2, 4)
1
```

## 13 四舍五入

四舍五入，第二个参数代表小数点后保留几位：

```
>>> round(10.045, 2)
10.04
>>> round(10.046, 2)
10.05
```

## 14 计算表达式

使用内置函数 `eval` 计算字符串型表达式的值：

```
>>> s = "1 + 3 + 5"
>>> eval(s)
9
>>> eval('[1, 3, 5]*3')
[1, 3, 5, 1, 3, 5, 1, 3, 5]
```

真假布尔值本质上也是数字，所以也归并到此节中讨论。

## 15 真假

以下四种情况都为假值：



```
>>> bool(0)
False
>>> bool(False)
False
>>> bool(None)
False
>>> bool([])
False
```

以下这些情况为真：

```
>>> bool([False])
True
>>> bool([0,0,0])
True
```

## 16 all 判断元素是否都为真

所有元素都为真返回 `True`，否则返回 `False`

```
#有0，所以不是所有元素都为真
>>> all([1,0,3,6])
False
#所有元素都为真
>>> all([1,2,3])
True
```

## 17 any 判断是否至少有一个元素为真

至少有一个元素为真返回 `True`，否则返回 `False`

```
# 没有一个元素为真
>>> any([0,0,0,[]])
False
# 至少一个元素为真
>>> any([0,0,1])
True
```

## 18 链式比较

Python 支持下面这种链式比较，非常方便：

```
>>> i = 3
>>> 1 < i < 3
False
>>> 1 < i <=3
True
```

## 19 交换元素

Python 除了支持上面的链式比较外，还支持一种更加方便的操作：直接解包赋值。

如下所示，1,3 解包后分别赋值给a,b，利用此原理一行代码实现两个数字的直接交换。

```
a, b = 1, 3
a, b = b, a # 交换 a,b
```

如果明白此原理，下面的赋值操作就会迎刃而解：

```
a,b = 1, 3
a, b = b+1, a-1
print(a,b) # 结果是多少？
```

可能会有疑问：是 `b+1` 赋值给 `a` 后，`a-1`再赋值给 `b` ？

如果明白了上面的原理：等号右面完成压包，左侧再解包。

就会立即得出答案：肯定不是。



下面这行代码：

```
a, b = b+1, a-1
```

等价于：

```
c = b+1, a-1 # 压包  
a, b = c # 解包
```

答案是：a=4, b=0

压包和解包还有更加复杂的用法，放到后面进阶部分总结。

## 20 链式操作

下面这个例子使用 `operator` 模块中 `add`, `sub` 函数，根据操作符 `+`, `-`，生成对应的函数，然后直接调用。

很像设计模式中最频繁使用的对象工厂模式。

```
>>> from operator import (add, sub)  
>>> def add_or_sub(a, b, oper):  
    return (add if oper == '+' else sub)(a, b)  
>>> add_or_sub(1, 2, '-')  
-1
```

## 二、Python 字符串专题

除了常见的数值型，字符串是另一种常遇到的类型。一般使用一对单引号或一对双引号表示一个字符串。

字符串中如果遇到 `\` 字符，可能是在做字符转义，所谓的转义便是字符的含义发生改变，比如常用的 `\n` 组合，转义后不再表示字符 `n` 本身，而是完成换行的功能。

类似的，还有很多转义字符，如 `\t`，正则表达式中 `\s`，`\d` 等等。

### 1 字符串创建

一般使用一对单引号或一对双引号表示一个字符串。如下所示 `s` 为字符串：

```
s = 'python' # 或 s = "python"
```

很多情况下单引号和双引号作用相同，但是一些情况还是存在微妙不同。

例如，使用一对双引号( `"` )时，打印下面字符串无需转义字符(也就是 `\` 字符)：

```
In [10]: print("That isn't a horse")
That isn't a horse
```

但是使用一对单引号打印时，却需要添加转义字符 `\`，如下所示：

```
In [11]: print('That isn\'t a horse')
That isn't a horse
```

除此之外，如果遇到字符串偏长，一行容不下，需要展示为多行。一对三重单引号 `'''` 或三重 `"""` 就会派上用场，它们能轻松实现跨行输入：

```
In [12]: print("""You're just pounding two
...: coconut halves together.""")
You're just pounding two
coconut halves together.
```

## 2 \ 转义

转义的语法：一个 `\` + 单个字符，组合后单个字符失去原来字面意义，会被赋予一个新的功能。

常见的转义字符：`\n` 完成换行，`\t` tab 空格等。

转义的另外一个重要作用是用于Python的正则。正则不仅指使用模块 `re` 完成字符串处理，还泛指很多常用包中函数的参数使用小巧的正则表达，比如数据分析必备包 `pandas`，`str` 访问器中 `split`，`cat` 等方法参数中使用转义字符。

关于正则处理字符串的常见用法，后面会有一个单独的专题总结。

## 3 字符串与数字

字符串与数字结合也会十分有用，可以玩出很多有趣的花样。

数字 `n` 乘以字符串会克隆出 `n` 倍个原字符串：

```
In [42]: 3*'py'
Out[42]: 'pypypy'
```

20乘以字符 - 会绘制出一条虚线：

```
In [43]: 20*'-'
Out[43]: '-----'
```

2个字符串常量能直接结合，中间不用添加任何东西，如下：

```
In [14]: 'Py'+'thon'
Out[14]: 'Python'
```

单个字符还能与数值完成互转，内置函数 `ord` 转换单个字符为整型，`chr` 函数转换整数为单个字符：

```
In [35]: ord('振')
Out[35]: 25391
In [36]: chr(25391)
Out[36]: '振'
```

还能使用 `bytes` 函数将字符串转为字节类型( `bytes` )，使用 `str` 函数转化字节类型为字符串：

```
s = 'python'
sa = bytes(s,encoding='utf-8')
s = str(sa,encoding='utf-8')
```

## 4 字符串打印及格式化

常规打印一个字符串比较简单，如果打印字符串中含有变量，该怎么正确打印。一般有两种方法：

## 1. 使用 `format` 函数

字符串变量使用一对花括号 `{}`，`format` 参数中指定变量的取值：

```
>>> print("i am {0},age {1}".format("tom",18))
i am tom,age 18
```

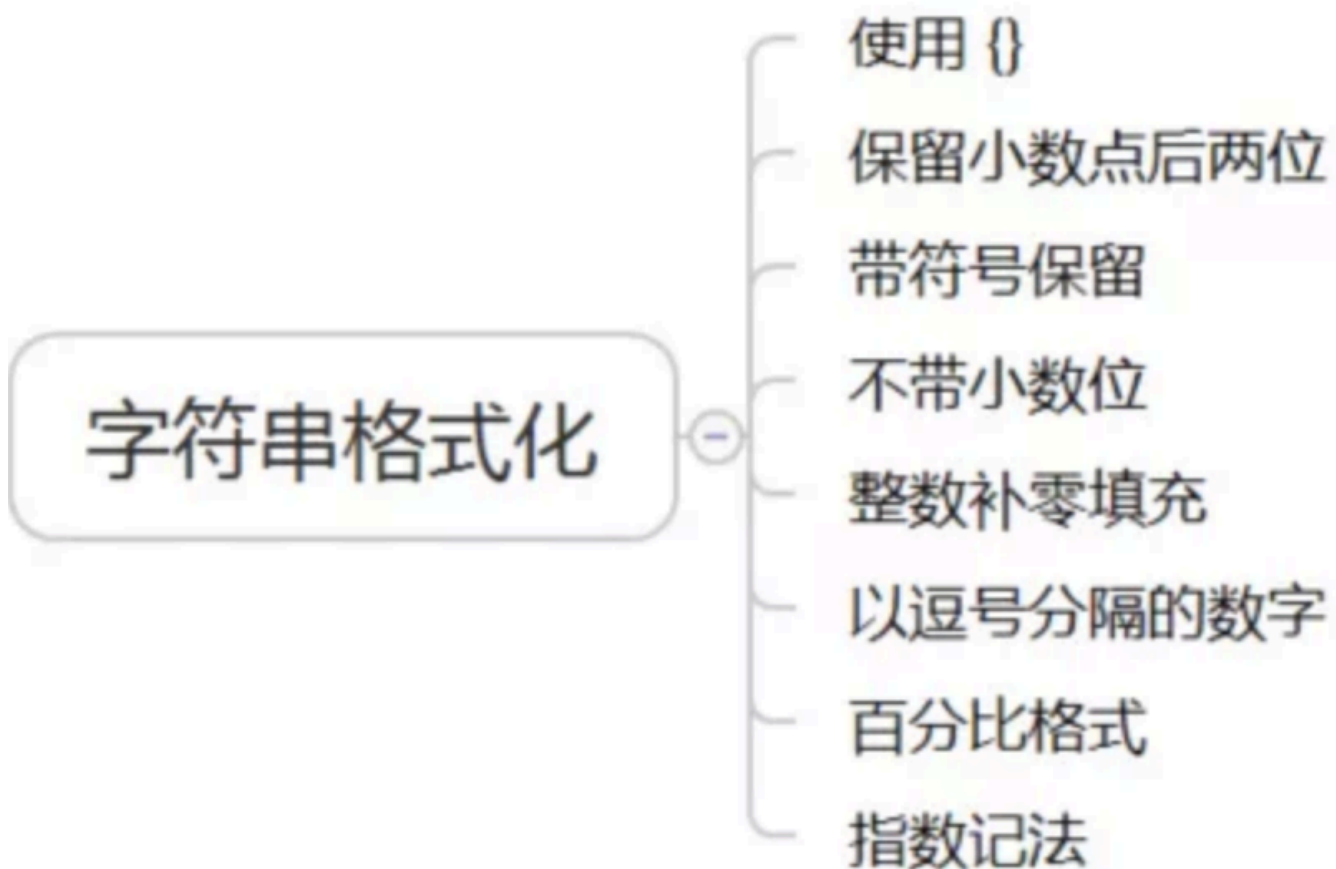
## 1. f 打印

f 后面紧跟一个字符串，其中花括号 {} 中直接写出变量名称，显然这种含有变量的打印方法更加符合大多数人的习惯：

```
In [45]: tom = 'tom'
In [46]: age = 18
In [47]: print(f'i am {tom}, age {age}')
i am tom, age 18
```

除了知道如何打印字符串，还有一项重要的事：如何控制字符串的打印。

虽然这是一个非常小的功能，但是知道一些常见的控制方法，却能使得书写更加简洁。常见的控制打印用法：



输出中如何控制保留两位小数，整数补零填充，对齐，百分比格式打印，整数太长使用科学计数法打印等等。可以记住如下**7种**常见用法：

(1). 保留小数点后两位

```
# 1 保留小数点后两位
>>> print("{:.2f}".format(3.1415926))
3.14
```

(2). 带符号保留小数点后两位

```
>>> print("{:+.2f}".format(-1))
-1.00
```

(3). 不带小数位

```
>>> print("{:.0f}".format(2.718)) # 不带小数位
3
```

(4). 整数补零，填充左边，宽度为3

```
>>> print("{:0>3d}".format(5)) # 整数补零，填充左边，宽度为3
005
```

(5). 以逗号分隔的数字格式

```
>>> print("{:,}".format(10241024)) # 以逗号分隔的数字格式
10,241,024
```

(6). 百分比格式

```
>>> print("{:.2%}".format(0.718)) # 百分比格式
71.80%
```

(7). 指数记法

```
>>> print("{:.2e}".format(10241024)) # 指数记法
1.02e+07
```

## 5 字符串常见处理操作

字符串对应Python中的 `str` 类型，其内置封装的方法有几十个，下面列举一些常用的必知的用法。

`join` 串联多个字符串，注意Python中没有单个字符这种类型，单个字符在Python中也会被当作 `str` 类型。如下连接多个字符串，最终打印出 `Python` 串：

```
chars = ['P', 'y', 't', 'h', 'o', 'n']
name = ''.join(chars)
```

既然有串联字符串，就应该有相反的操作：分割字符串，一般使用 `split` 函数，第一个参数指明分割字符串使用的分割符：

```
In [49]: 'col1,col2,col3'.split(',')
Out[49]: ['col1', 'col2', 'col3']
```

`split` 还有第二个参数指明需要做的分割次数，比如只做一次分割，得到如下两个元素：

```
In [51]: 'col1,col2,col3'.split(',',1)
Out[51]: ['col1', 'col2,col3']
```

`split` 默认是从左侧开始分割字符串，与之对应的另一个函数 `rsplit` 就是从右侧开始分割字符串，某些场景 `rsplit` 函数更好用一些。从右侧开始只做一次分割可以写为：

```
In [52]: 'col1,col2,col3'.rsplit(',',1)
Out[52]: ['col1,col2', 'col3']
```

除了以上两个常用的方法，还有 `replace` , `startswith` , `strip` , `rstrip` 等等经常也会用到，在此不再一一举例。

以上就是字符串处理的基本用法专题，主要总结了：

- 1 字符串创建
- 2 \转义
- 3 字符串与数字
- 4 字符串打印及格式化
- 5 字符串常见处理操作

最后，以一个更有意思的小功能作为本专题的收尾。

已知下面一个长句，将其转化为多行，每行只有11个字符。

```
words = '是想与朋友们分享一个再普通不过的道理：脚踏实地做些实事，哪怕是不起眼的小事，每天前进一点，日积月累会做出一点成绩的。'
```

借助内置的 `textwrap` 模块中 `fill` 方法，实现每行 11 个字符：

```
import textwrap
r = textwrap.fill(words, 11)
print(r)
```

结果：

```
是想与朋友们分享一个再
普通不过的道理：脚踏实
地做些实事，哪怕是不起
眼的小事，每天前进一点
，日积月累会做出一点成
绩的。
```

## 三、列表专题

### 1 创建列表

列表是一个容器，使用一对中括号 `[]` 创建一个列表。

创建一个空列表：

```
a = [] # 空列表
```

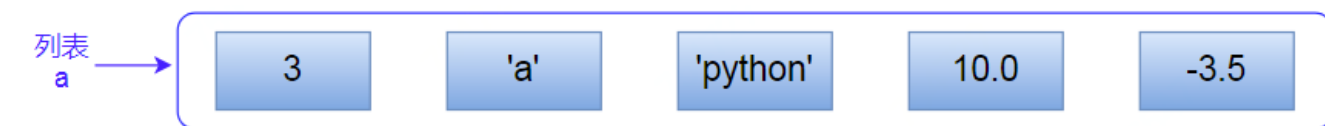
创建一个含有 5 个整型元素的列表 `a`：

```
a = [3, 7, 4, 2, 6]
```

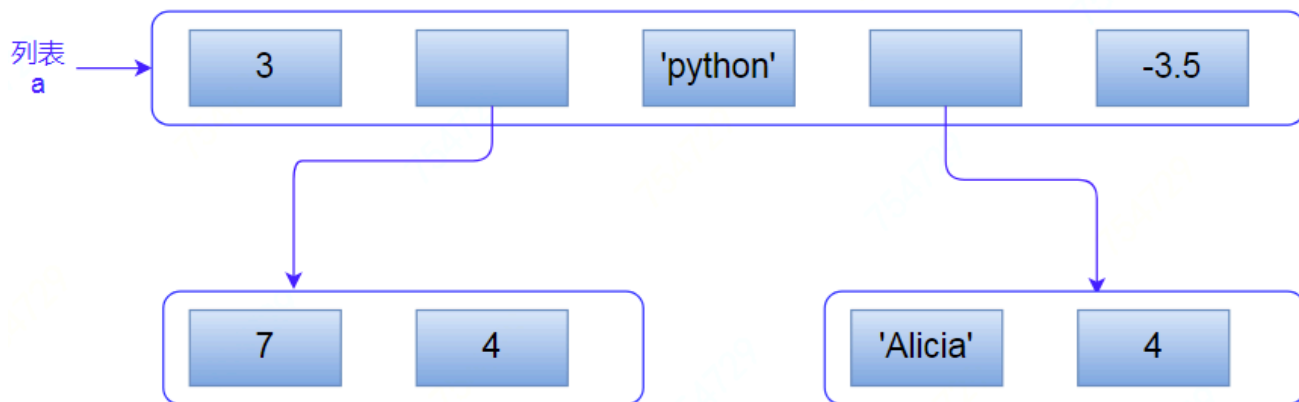
列表与我们熟知的数组很相似，但又有很大区别。

一般数组内的元素要求同一类型，但是列表内可含有各种不同类型，包括再嵌套列表。

如下，列表 `a` 包含三种类型：整形，字符串，浮点型：



如下列表 `a` 嵌套两个列表：



## 2 访问元素

列表访问主要包括两种：索引和切片。

如下，访问列表 `a` 可通过我们所熟知的正向索引，注意从 `0` 开始；

列表 a	3	7	4	2	6
正索引	0	1	2	3	4
负索引	-5	-4	-3	-2	-1

也可通过Python特有的负向索引，

即从列表最后一个元素往前访问，此时索引依次被标记为 `-1, -2, ..., -5`，注意从 `-1` 开始。

除了以上通过索引访问单个元素方式外，



还有非常像 `matlab` 的切片访问方式，这是一次访问多个元素的方法。

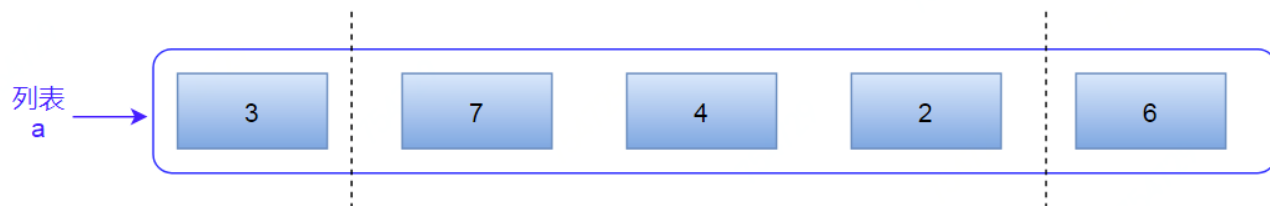
切片访问的最基本结构：中间添加一个冒号。

如下切片，能一次实现访问索引为1到4，不包括4的序列：

```
In [1]: a=[3,7,4,2,6]
```

```
In [2]: a[1:4]
```

```
Out[2]: [7, 4, 2]
```



Python支持负索引，能带来很多便利。比如能很方便的获取最后三个元素：

```
In [1]: a=[3,7,4,2,6]
```

```
In [3]: a[-3:]
```

```
Out[3]: [4, 2, 6]
```

除了使用一个冒号得到连续切片外，

使用两个冒号获取带间隔的序列元素，两个冒号后的数字就是间隔长度：

```
In [1]: a=[3,7,4,2,6]
```

```
In [7]: a[::2] # 得到切片间隔为2
```

```
Out[7]: [3, 4, 6]
```

其实最全的切片结构：`start:stop:interval`，如下所示，获得切片为：索引从1到5间隔为2：

```
In [6]: a=[3,7,4,2,6]
```

```
In [7]: a[1:5:2]
```

```
Out[7]: [7, 2]
```

### 3 添加元素

列表与数组的另一个很大不同，使用数组前，需要知道数组长度，便于从系统中申请内存。

但是，列表却不需要预先设置元素长度。

它支持任意的动态添加元素，完全不用操心列表长短。

它会随着数组增加或删除而动态的调整列表大小。

这与数据结构中的线性表或向量很相似。

添加元素通常有两类场景。

`append` 一次添加1个元素，`insert` 在指定位置添加元素：

```
In [8]: a=[3,7,4,2,6]
In [9]: a.append(1) # append默认在列表尾部添加元素
In [10]: a
Out[10]: [3, 7, 4, 2, 6, 1]

In [11]: a.insert(2,5) # insert 在索引2处添加元素5
In [12]: a
Out[12]: [3, 7, 5, 4, 2, 6, 1]
```

`extend` 或直接使用 `+` 实现一次添加多个元素：

```
In [13]: a.extend([0,10])# 一次就地添加0,10两个元素
In [14]: a
Out[14]: [3, 7, 5, 4, 2, 6, 1, 0, 10]

In [15]: b = a+[11,21] # + 不是就地添加，而是重新创建一个新的列表
In [16]: b
Out[16]: [3, 7, 5, 4, 2, 6, 1, 0, 10, 11, 21]
```

这里面有一个重要细节，不知大家平时注意到吗。

`extend` 方法实现批量添加元素时未创建一个新的列表，而是直接添加在原列表中，这被称为 `in-place`，就地。而 `b=a+list` 对象实际是创建一个新的列表对象，所以不是就地批量添加元素。

但是，`a+=` 一个列表对象，`+=` 操作符则就会自动调用 `extend` 方法进行合并运算。大家注意这些

微妙的区别，不同场景选用不同的API，以此高效节省内存。

## 4 删除元素

删除元素的方法有三种：`remove`，`pop`，`del`。

`remove` 直接删除元素，若被删除元素在列表内重复出现多次，则只删除第一次：

```
In [17]: a=[1,2,3,2,4,2]
In [18]: a.remove(2)
In [19]: a
Out[19]: [1, 3, 2, 4, 2]
```

`pop` 方法若不带参数默认删除列表最后一个元素；若带参数则删除此参数代表的索引处的元素：

```
In[19]: a = [1, 3, 2, 4, 2]
In [20]: a.pop() # 删除最后一个元素
Out[20]: 2
In [21]: a
Out[21]: [1, 3, 2, 4]

In [22]: a.pop(1) # 删除索引等于1的元素
Out[22]: 3
In [23]: a
Out[23]: [1, 2, 4]
```

`del` 与 `pop` 相似，删除指定索引处的元素：

```
In [24]: a = [1, 2, 4]
In [25]: del a[1:] # 删除索引1到最后的切片序列
In [26]: a
Out[26]: [1]
```

## 5 list 与 in

列表是可迭代的，除了使用类似 `c` 语言的索引遍历外，还支持 `for item in alist` 这种直接遍历元素的方法：

```
In [28]: a = [3,7,4,2,6]
In [29]: for item in a:
...:     print(item)
3
7
4
2
6
```

`in` 与可迭代容器的结合，还用于判断某个元素是否属于此列表：

```
In [28]: a = [3,7,4,2,6]
In [30]: 4 in a
Out[30]: True

In [31]: 5 in a
Out[31]: False
```

## 6 list 与数字

内置的list与数字结合，实现元素的复制，如下所示：

```
In [32]: ['Hi!'] * 4
Out[32]: ['Hi!', 'Hi!', 'Hi!', 'Hi!']
```

表面上这种操作太方便，实际确实也很方便，比如我想快速打印20个 `-`，只需下面一行代码：

```
In [33]: '-' * 20
Out[33]: '-----'
```

使用列表与数字相乘构建二维列表，然后第一个元素赋值为 `[1,2]`，第二个元素赋值为 `[3,4]`，第三个元素为 `[5]`：

```
In [34]: a = [[]] * 3
In [35]: a[0]=[1,2]
In [36]: a[1]=[3,4]
In [37]: a[2]=[5]
In [38]: a
Out[38]: [[1, 2], [3, 4], [5]]
```

## 7 列表生成式

列表生成式 是创建列表的一个方法，它与使用 `append` 等API创建列表相比，书写更加简洁。

使用列表生成式创建1到50的所有奇数列表：

```
a=[i for i in range(50) if i&1]
```

## 8 其他常用API

除了上面提到的方法外，列表封装的其他方法还包括如下：

```
clear` , `index` , `count` , `sort` , `reverse` , `copy
```

`clear` 用于清空列表内的所有元素 `index` 用于查找里面某个元素的索引：

```
In [4]: a=[1,3,7]
```

```
In [5]: a.index(7)
```

```
Out[5]: 2
```

`count` 用于统计某元素的出现次数：

```
In [6]: a=[1,2,3,2,2,5]
```

```
In [7]: a.count(2) # 元素2出现3次
```

```
Out[7]: 3
```

`sort` 用于元素排序，其中参数 `key` 定制排序规则。如下列表，其元素为元祖，根据元祖的第二个值由小到大排序：

```
In [8]: a=[(3,1),(4,1),(1,3),(5,4),(9,-10)]
```

```
In [9]: a.sort(key=lambda x:x[1])
```

```
In [10]: a
```

```
Out[10]: [(9, -10), (3, 1), (4, 1), (1, 3), (5, 4)]
```

`reverse` 完成列表反转：

```
In [15]: a=[1, 3, -2]

In [16]: a.reverse()

In [17]: a
Out[17]: [-2, 3, 1]
```

`copy` 方法在下面讲深浅拷贝时会详细展开。

## 9 列表实现栈

列表封装的这些方法，实现 `栈` 这个常用的数据结构比较容易。栈是一种只能在列表一端进出的特殊列表，`pop` 方法正好完美实现：

```
In [23]: stack=[1, 3, 5]

In [24]: stack.append(0) # push元素0到尾端，不需要指定索引

In [25]: stack
Out[25]: [1, 3, 5, 0]

In [26]: stack.pop() # pop元素，不需指定索引，此时移出尾端元素
Out[26]: 0

In [27]: stack
Out[27]: [1, 3, 5]
```

由此可见Python的列表当做栈用，完全没有问题，`push` 和 `pop` 操作的时间复杂度都为  $O(1)$

但是使用列表模拟队列就不那么高效了，需要借助Python的 `collections` 模块中的双端队列 `deque` 实现。

## 10 列表包含自身

列表的赋值操作，有一个非常有意思的问题，大家不妨耐心看一下。

```
In [1]: a=[1, 3, 5]

In [2]: a[1]=a # 列表内元素指向自身
```

这样相当于创建了一个引用自身的结构。

打印结果显示是这样的：

```
In [3]: a
Out[3]: [1, [...], 5]
```

中间省略号表示无限循环，这种赋值操作导致无限循环，这是为什么？下面分析下原因。

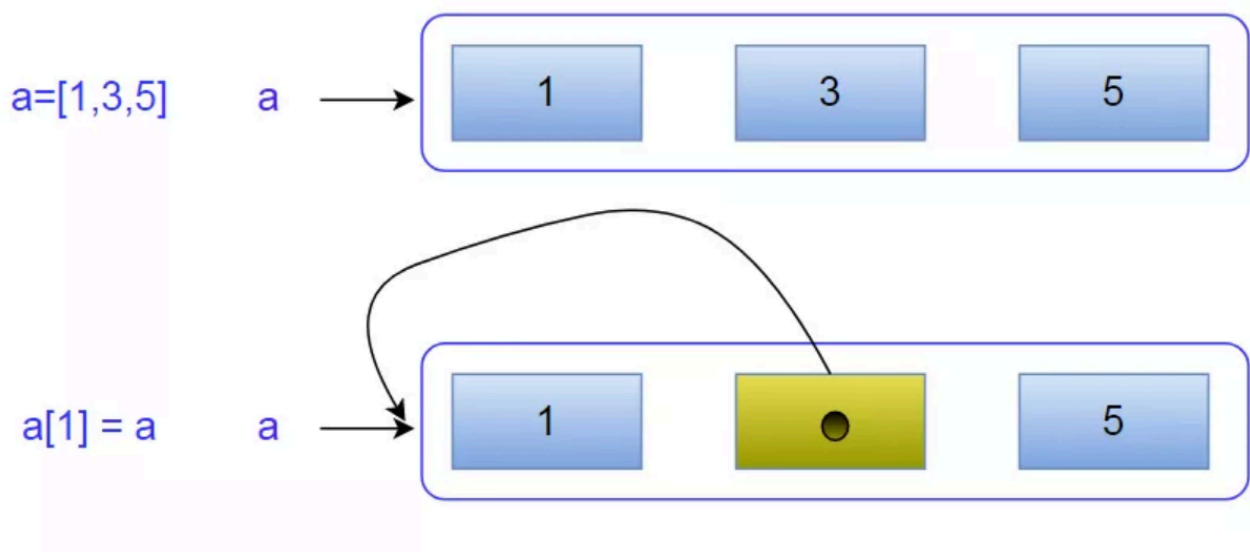
执行 `a = [1,3,5]` 的时候，Python 做的事情是首先创建一个列表对象 `[1, 3, 5]`，然后给它贴上名为 `a` 的标签。

执行 `a[1] = a` 的时候，Python 做的事情则是把列表对象的第二个元素指向 `a` 所引用的列表对象本身。

执行完毕后，`a` 标签还是指向原来的那个对象，只不过那个对象的结构发生了变化。

从之前的列表 `[1,3,5]` 变成了 `[1,[...], 5]`，而这个`[...]`则是指向原来对象本身的一个引用。

如下图所示：



可以看到形成一个环路：`a[1]---`中间元素`--->a[1]`，所以导致无限循环。

## 11 插入元素性能分析

与常规数组需要预先指定长度不同，Python 中list不需要指定容器长度，允许我们随意的添加删除元素。

但是这种便捷性也会带来一定副作用，就是插入元素的时间复杂度为 $O(n)$ ，而不是 $O(1)$ ，因为`insert`会导致依次移动插入位置后的所有元素。

为了加深对插入元素的理解，特意把cpython实现 `insert` 元素的操作源码拿出来。

可以清楚看到 `insert` 元素时，插入位置处的元素都会后移一个位置，因此插入元素的时间复杂度为  $O(n)$ ，所以凡是涉及频繁插入删除元素的操作，都不太适合用 `list`。

```
static int
ins1(PyListObject *self, Py_ssize_t where, PyObject *v)
{
    assert((size_t)n + 1 < PY_SSIZE_T_MAX);
    if (list_resize(self, n+1) < 0)
        return -1;

    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    items = self->ob_item;
    //依次移动插入位置后的所有元素
    // O(n) 时间复杂度
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
    items[where] = v;
    return 0;
}
```

## 12 深浅拷贝

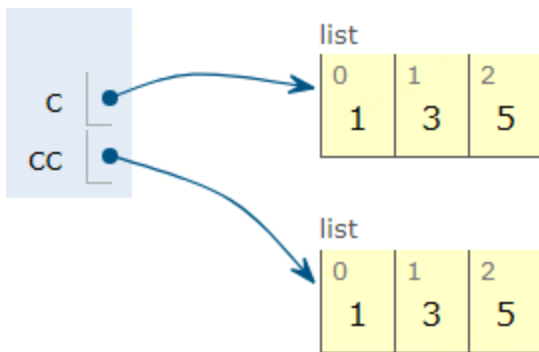
`list` 封装的 `copy` 方法实现对列表的浅拷贝，浅拷贝只拷贝一层，具体拿例子说：

```
In [38]: c =[1,3,5]

In [39]: cc = c.copy()
```

`c` 和 `cc` 分别指向一片不同内存，示意图如下：





这样修改 `cc` 的第一个元素，原来 `c` 不受影响：

```
In [40]: cc[0]=10 # 修改cc第一个元素
```

```
In [41]: cc
Out[41]: [10, 3, 5]
```

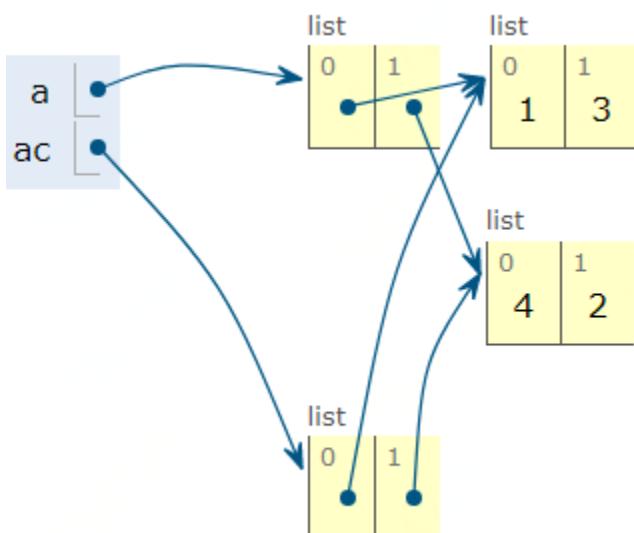
```
In [42]: c # 原来 c 不受影响
Out[42]: [1, 3, 5]
```

但是，如果内嵌一层列表，再使用 `copy` 时只拷贝一层：

```
In [32]: a=[[1,3],[4,2]]
```

```
In [33]: ac = a.copy()
```

```
In [34]: ac
Out[34]: [[1, 3], [4, 2]]
```



上面的示意图清晰的反映出这一点，内嵌的列表并没有实现拷贝。因此再修改内嵌的元素时，原来的列表也会受到影响。

```
In [35]: ac[0][0]=10

In [36]: ac
Out[36]: [[10, 3], [4, 2]]

In [37]: a
Out[37]: [[10, 3], [4, 2]]
```

要想实现深度拷贝，需要使用Python模块 `copy` 中的 `deepcopy` 方法。

## 13 列表可变性

列表是可变的，可变的对象是不可哈希的，不可哈希的对象不能被映射，因此不能被用作字典的键。

```
In [51]: a=[1,3]
In [52]: d={a:'不能被哈希'} #会抛出如下异常

# TypeError: unhashable type: 'list'
```

但是，有时我们确实需要列表对象作为键，这怎么办？

可以将列表转化为元祖，元祖是可哈希的，所以能作为字典的键。

## 四、流程控制专题

流程控制与代码的执行顺序息息相关，流程控制相关的关键字，如 `if`，`elif`，`for`，`while`，`break`，`continue`，`else`，`return`，`yield`，`pass` 等。

本专题详细总结与流程控制相关的基础和进阶用法，大纲如下：

- 基础用法
  - 1 if 用法
  - 2 for 用法
  - 3 while,break,continue
- 进阶用法

- - 4 for 使用注意
  - 5 range 序列
  - 6 Python特色：循环与else
  - 7 pass 与接口
  - 8 return 和 yield
  - 9 短路原则

专题的开始，先总结与流程控制相关的基础用法。

## 1 if 用法

`if` 对应逻辑控制的条件语句，它的基本结构可以表示为：如果满足某个条件，则怎么怎么样。

如下函数 `maxChunksToSort` 中，如果满足当前数组 `nums` 的索引 `i` 等于区间 `[0,i]` 的最大值，则 `[0,i]` 区间能被分割为一个Chunk.

```
def maxChunksToSort(nums):
    maxn, count = nums[0], 0
    for i, num in enumerate(nums):
        maxn = max(maxn, num)
        if i == maxn:
            count += 1
    return count
```

`if` 后的语句指定了一个条件，若满足 `if` 则，`:` 后的语句成立。

如果 `if` 不满足，再使用 `elif` 判断其他情况，可以一直写 `elif`，若是最后一个判断条件，可使用 `else`，其基本结构为：

```
if A:
    print('condition A meets')
elif B:
    print('condition B meets')
elif C:
    print('condition C meets')
else:
    print('other conditions meets')
```

## 2 for 用法

Python的 `for` 除了具备控制循环次数外，还能直接迭代容器中的元素。

控制循环次数：

```
for i in range(1, len(nums)):
    print(i)
```

还能直接操作容器内的元素：

```
a = [1, [2, 4], [5, 7]]
for item in a:
    print(item)
```

## 3 while,break,continue

`while` 后面紧跟一个判断条件，若满足条件则会一直循环，直到不满足条件时退出。但这不是绝对的，如果`while`后的语句块内含有 `break`，即便条件依然满足，但遇到 `break` 也会一样退出。

如下检测输入是否为整数，直到输入整数时，执行 `break` 退出 `while` 循环：

```
while True:
    a = input('please input an Integer: ')
    try:
        ai = int(a)
        print('输入了一个整数 %d , input 结束' % (ai,))
        break
    except:
        print("%s isn't a Integer" % (a,))
```

做如下测试：

```
please input an Integer: 1.2
1.2 isn't a Integer
please input an Integer: 1
输入了一个整数 1 , input 结束
```

`continue` 与最近的循环语句 `for` 或 `while` 组合，表示接下来循环体内的语句不执行，重新进

入下一次遍历。

```
def f(nums):  
    for num in nums:  
        if num <= 0:  
            continue  
        print('得到一个大于0的数 %d' % (num,))
```

做如下测试：

```
得到一个大于0的数 2  
得到一个大于0的数 4
```

基础用法保证我们能够应付日常遇到的基本的代码流程，不过要想进一步深入理解Python特色的、与顺序相关的执行功能，还需要理解下面的进阶用法。

## 4 for 使用注意

for 语句遍历容器类型或可迭代类型时，如果涉及到增加、删除元素，就需要小心。比如请先看下面的例子：

删除列表中的某个元素值，可能有重复，要求元素顺序不变，空间复杂度为 $O(1)$ ，如果像下面这样写就会有问题：

```
def delItems(nums, target):  
    for item in nums:  
        if item == target:  
            nums.remove(item)  
    return nums
```

对于大多数情况，上面的代码无法暴露出bug。但是考虑下面输入(特点：被删除的值连续出现)：

```
r = delItems([2, 1, 3, 1, 1, 3], 1)  
print(r)
```

打印结果为：`[2,3,1,3]`

对于刚接触编程的朋友对此很不解，为什么其中一个1未被remove.


不管是Python, Java, C++，列表或数组删除元素时，其后面的元素都会逐次前移1位，但是 `for` 依然会正常迭代，因此“成功”规避了相邻的后面元素1.

图形解释命中目标后的一系列动作：

值	2	1	3	1	1	3
索引	0	1	2	3	4	5

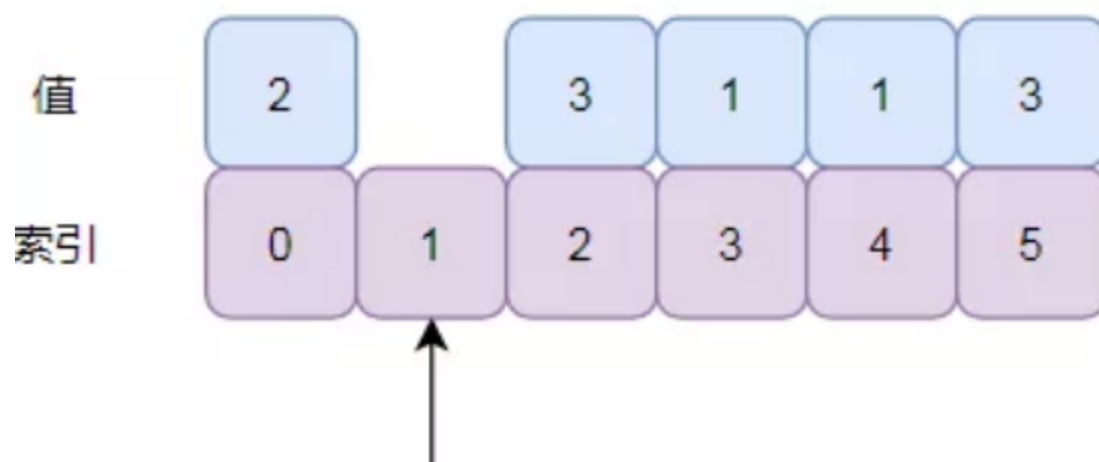
上面的列表

	命中目标					
值	2	1	3	1	1	3
索引	0	1	2	3	4	5



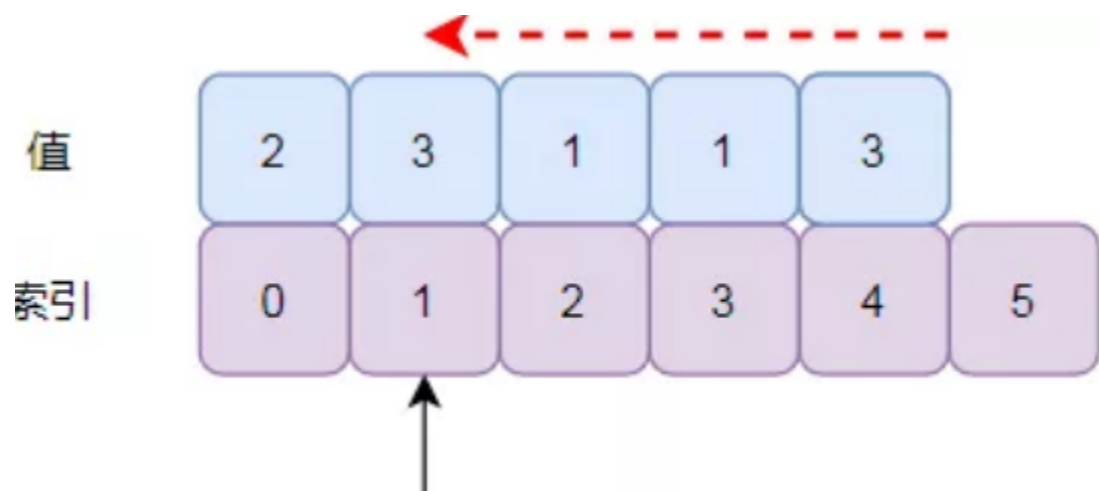
命中目标

删除1

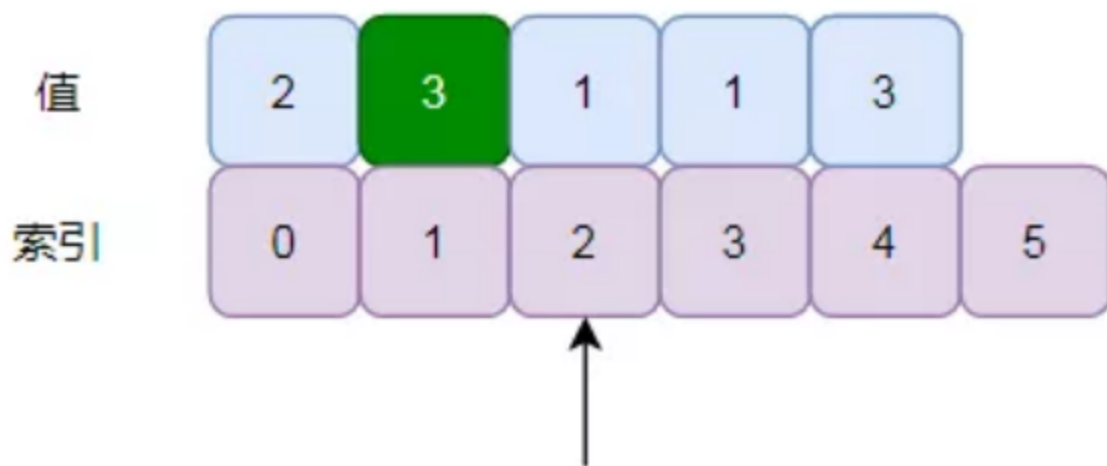


删除元素1

下步最关键：解释器自动前移删除位置后的所有元素



但是，等到下一次迭代时，迭代器不等待，正常移动到下一个位置：



这样元素 3 成功逃避是否与目标值相等的检查。

**结论：**命中目标处的后一个位置都逃避了是否与目标值相等的检查，所以一旦有连续目标值，必然就会漏掉，进而触发上面的bug.

明白上面这个原因后，重新再改写一遍删除所有重复元素的代码，下面代码不再使用 `for` 直接遍历元素(再说一遍：增删元素原来迭代器发生改变，所以会导致异常行为)，而是使用索引访问：

```
def delItems(nums, target):
    i = 0
    while i < len(nums):
        if nums[i] == target:
            del nums[i]
            i -= 1
        i += 1
    return nums

r = delItems([2, 1, 3, 1, 1, 3], 1)
print(r) # [2, 3, 3]
```

如果元素等于 `target`，从数组 `nums` 中删除 `nums[i]`，删除后解释器自动将 `i` 后的元素都前移 1 位。据此，巧妙的控制 `i` 值，一旦命中立即 `i` 减去 1，这样确保不漏检查。

## 5 range 序列

`range` 在 Python 中经常用于生成一串数字序列，对刚入门 Python 的朋友想尝试打印其中的值：



```
In [3]: print(range(10))
range(0, 10)
```

要想看到每个值可与 `for` 结合：

```
In [21]: for i in range(10):
...:     print(i,end=",")
...:
0,1,2,3,4,5,6,7,8,9,
```

那么有的朋友不禁要问 `range` 函数的返回值为什么能与 `for` 结合？

类型为 `Iterable` 的对象都可与 `for` 结合，下面确认 `range(10)` 返回值是否为 `Iterable`：

```
In [13]: from collections.abc import Iterable
In [14]: isinstance(range(10),Iterable)
Out[14]: True # 它是 Iterable 类型
```

它为什么不是一次全部输出一个列表，就像下面这样：

```
In [23]: list(range(10))
Out[23]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

而是要一个一个的输出？

`range` 函数为了高效节省内存，一次只返回一个值，而不是直接将构成序列的全部元素加载到内存。

Python里的`range`不支持创建浮点序列，所以为了更加清楚的展示 `range` 的原理，编写一个创建浮点数的序列 `frange`：

```
def frange(start, stop, step):
    i = start
    while i < stop:
        yield i
        i += step
```

代码只有几行，`yield` 作为控制流程的一个关键字，下面我们会详细说到。

使用 `frange`：

```
fr = frange(0, 1., 0.2)
for i in fr:
    print("{:.2f}".format(i), end=", ")
```

打印结果如下，得到一个差值为 0.2 的等差数列：

```
0.00, 0.20, 0.40, 0.60, 0.80,
```

## 6 Python特色：循环与else

### 6.1 for 能和 else 组对

Python一大特色：while, for 能和 else 组对，不仅如此，try except 和 else 也能组对，下面介绍它们存在的价值。

找出2到15的所有素数，如果不是素数打印出一对因子，实现代码如下：

```
for num in range(2, 16):
    is_prime = True
    for item in range(2, num):
        if num % item == 0:
            print('%d = %d*%d ' % (num, item, num // item))
            is_prime = False
            break
    if is_prime:
        print("%d is prime" % (num))
```

打印结果如下：

```
2 is prime
3 is prime
4 = 2*2
5 is prime
6 = 2*3
7 is prime
8 = 2*4
9 = 3*3
10 = 2*5
11 is prime
12 = 2*6
13 is prime
14 = 2*7
15 = 3*5
```

使用 `is_prime` 标志位判断是否找到 `num` 的一对因子，若都遍历完仍无发现则打印此数是素数。

这是我们比较熟悉的常规解决思路，但是如果使用 `for` 和 `else` 组对，它的价值便能体现出来：

```
for num in range(2, 16):
    for item in range(2, num):
        if num % item == 0:
            print('%d = %d*%d ' % (num, item, num // item))
            break
    else:
        print("%d is prime" % (num))
```

上面代码实现同样的功能，但代码相对更加简洁。通过前后代码对比，我们便能看出 `for` 和 `else` 组对的功能：`for` 遍历完成后执行 `else`，但是触发 `break` 后，`else` 不执行。

大家平时多多使用，便能习惯以上用法。通过上面的对比，我们也能直观地感受到它们的价值。

## 6.2 `try, except` 和 `else` 组对

`try` 和 `except` 组对比较容易理解，触发异常执行 `except` 里的代码，否则不执行。

但是加上一个 `else` 实现怎样的作用呢？

首先看下面的例子：

```
In [9]: while True:
...:     try:
...:         a = int(input('请输入一个整数: '))
...:     except ValueError:
...:         print('input value is not a valid number')
...:     else:
...:         if a % 2 == 0:
...:             print('输入的 %d 是偶数' %(a,))
...:         else:
...:             print('输入的 %d 是奇数' %(a,))
...:         break
```

测试：

```
请输入一个整数: t
input value is not a valid number
请输入一个整数: 5
输入的 5 是奇数
```

try 保护的代码正常通过后，else 才执行。

有的朋友会问，为什么不把 else 这块代码放到 try 里面？这还是有一定区别的：放到 else 中意味着这块代码不必受保护，因为它不可能触发 ValueError 这样的异常。

## 7 pass 与接口

Python中最特别的关键字之一便是 pass，它放在类或函数里，表示类和函数暂不定义。

```
class PassClass:
    pass

def PassFun():
    pass
```

如上实现最精简的类和函数定义。

今天跟大家分享一个 pass 的特别有用的用法，尤其对 Java 语言的 interface，implements 等较熟悉的朋友，在Python中也能实现类似写法。

首先安装一个包:

```
pip install python-interface
```

下面是这个包的基本用法，首先创建一个接口类：

```
from interface import implements, Interface

class MyInterface(Interface):

    def method1(self, x):
        pass

    def method2(self, x, y):
        pass
```

下面实现接口：

```
class MyClass(implements(MyInterface)):

    def method1(self, x):
        return x * 2

    def method2(self, x, y):
        return x + y
```

这个包对熟悉 `Java` 语言的朋友还是非常实用，接口和实现类用法，可以平稳过渡到 `Python` 语言中。

## 8 return 和 yield

程序遇到 `return` 和 `yield` 都是立即中断返回。那么 `yield` 和 `return` 又有什么不同呢？

与 `return` 不同，`yield` 中断返回后，下一次迭代会进入到 `yield` 后面的下一行代码，而不像 `return` 下一次执行还是从函数体的第一句开始执行。

用图解释一下：

```
>>> def fibonacci(n):  
    a, b = 1, 1  
    for _ in range(n):  
        yield a  
        a, b = b, a+b # 注意这种赋值  
  
>>> for fib in fibonacci(10):  
    print(fib)
```

遇到yield中断返回

第一次 yield 返回 1

```
>>> def fibonacci(n):  
    a, b = 1, 1  
    for _ in range(n):  
        yield a  
        a, b = b, a+b  
  
>>> for fib in fibonacci(10):  
    print(fib)
```

第二次迭代，直接到位置2这句代码。

然后再走 for ,再 yield ，重复下去，直到 for 结束。

以上就是理解 `yield` 的一个重要点，当然 `yield` 还会与 `from` 连用，还能与 `send` 实现协程等，这些都放在后面的专题。

## 9 短路原则

最后以一个有意思的短路问题结束流程控制专题。

布尔运算符 `and` 和 `or` 也被称为**短路**运算符：它们的参数从左至右解析，一旦可以确定结果解析就会停止。

Python中的短路运算符常见的有两个：`and`，`or`

A and B: 如果 A 不成立，B 不会执行

A or B: 如过 A 成立，B不会执行

所以被称为短路运算符

举几个例子一看就明白，请看下面代码：

代码1：

```
a = ''
b = a and 'i will not execute'
print(b)
```

打印结果为空，因为 `and` 运算符从左到右检查，一旦 `a` 为空即为假，则结果已确定为假，`'i will not execute'` 被短路。

代码2：

```
a = 'python'
b = a or 'i will not execute'
print(b)
```

打印结果为：`python`，因为 `or` 运算符从左到右检查，一旦 `a` 为真则结果已确定为真，所以 `'i will not execute'` 被短路。

## 五、Python编程习惯专题

今天讨论 Python 编程风格，如何写出更加Pythonic的代码是本篇讨论的话题。

Python代码的编程习惯主要参考 PEP8：

<https://www.python.org/dev/peps/pep-0008/>

里面主要包括如每行代码长度不超过79，函数间空一行等。

其实这些格式化的东西，现有的工具能够辅助我们很快满足编程风格，如 flake8 等小插件。

所以，这篇专题总结不会过多去讲语法相关的格式化，更多精力放在对比分析上，告诉大家常用的代码书写习惯，哪些写法不够符合习惯等。

### 2 多余的空格

函数赋值以下符合习惯：

```
foo(a, b=0, {'a':1, 'b':2}, (10,))
```

但是，下面多余空格不符合习惯：

```
# 这些空格都是多余的
foo ( a, b = 0, { 'a':1, 'b':2 }, (10, ))
```

但是下面代码，有空格又是更符合习惯的：

```
i += 1
num = num**2 + 1
def foo(nums: List)
```

尤其容易忽略的一个空格，增加函数元信息时要有一个空格：

```
def foo(nums: list): # 此处根据官方建议nums: list间要留有一个空格
    pass
```



## 1.2 是否为 None 判断

判断某个对象是否为 `None`，下面符合习惯：

```
if arr is None:
    pass

if arr is not None:
    pass
```

下面写法不符合习惯，一般很少见：

```
if arr == None:
    pass
```

特别的，对于 `list`，`tuple`，`set`，`dict`，`str` 等对象，使用下面方法判断是否为 `None` 更加符合习惯：

```
if not arr: #为 None 时，满足条件
    pass

if arr: # 不为 None 时，满足条件
    pass
```

## 3 lamda 表达式

lambda 表达式适合一些key参数赋值等，一般不习惯这么写：

```
f = lambda i: i&1
```

下面写法更加符合习惯：

```
def is_odd(i): return i&1
```

## 4 最小化受保护代码

要想代码更健壮，我们一般都做防御性的工作，最小化受保护的代码更加符合习惯，如下为了防御键不存在问题，加一个try:

```
try:
    val = d['c']
except KeyError:
    print('c' not existence)
```

上面写法是合理的，但是下面代码在捕获`KeyError`时，又嵌套一个函数是不符合习惯的：

```
try:
    val = foo(d['c']) # 这样写也会捕获foo函数中的KeyError异常
except KeyError:
    print('c' not existence)
```

这样写也会捕获`foo`函数中的`KeyError`异常，不符合习惯。

## 5 保持逻辑完整性

根据官方指南，只有 `if` 逻辑`return`，而忽视可能的 `x` 为负时的 `else` 逻辑，不可取：

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)
```

建议写法：

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None
```

或者这样写：

```
def foo(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

所以，不要为了刻意追求代码行数最少，而忽视使用习惯。

## 6 使用语义更加明确的方法

判断字符串是否以 `ize` 结尾时，不建议这样写：

```
if s[-3:] == 'ize':  
    print('ends ize')
```

使用字符串的 `endswith` 方法判断是否以什么字符串结尾，显然可读性更好：

```
if s.endswith('ize'):  
    print('ends ize')
```

以上这些只要平时多加注意，理解起来不是问题。其实除了 *PEP8* 指定的这些代码编写习惯外，还有一种与代码健壮性息息相关的编程风格，今天重点介绍这方面的编程习惯。

## 7 EAFP 防御编程风格

为了提升代码的健壮性，我们要做防御性编程，Python 中的 `try` 和 `except` 就是主要用来做这个：

```
d = {'a': 1, 'b': [1, 2, 3]}  
  
try:  
    val = d['c']  
except KeyError:  
    print('key not existence')
```

`try` 块中代码是受保护的，如果键不存在，`except` 捕获到 `KeyError` 异常，并处理这个异常信息。

而下面的代码，一旦从字典中获取不存在的键，如果没有任何 `try` 保护，则程序直接中断在这里，表现出来的现象就是 app 直接挂掉或闪退，这显然非常不友好。

```
d = {'a': 1, 'b': [1, 2, 3]}  
val = d['c']
```

再举一个 `try` 和 `except` 使用的例子，如果目录已存在则触发 `OSError` 异常，并通过 `except` 捕获到然后在块里面做一些异常处理逻辑。

```
import os
try:
    os.makedirs(path)
except OSError as exception:
    if exception.errno != errno.EEXIST:
        raise # PermissionError 等异常
    else:
        # path 目录已存在
```

以上这种使用 `try` 和 `except` 的防御性编程风格，在Python中有一个比较抽象的名字：**EAFP**

它的全称为：

*Easier to Ask for Forgiveness than Permission.*

没必要纠结上面这句话的哲学含义。

知道在编程方面的指代意义就行：首先相信程序会正确执行，然后如果出错了我们再处理错误。

使用 `try` 和 `except` 这种防御风格，优点明显，`try`里只写我们的业务逻辑，`except`里写异常处理逻辑，几乎无多余代码，Python指南里也提倡使用这种风格。

但是任何事物都有两面性，这种写法也不例外。那么，EAFP防御风格有何问题呢？它主要会带来一些我们不想出现的副作用。

举一个例子，如下`try`块里的逻辑：出现某种情况修改磁盘的csv文件里的某个值，这些逻辑都顺利完成，但是走到下面这句代码时程序出现异常，进而被 `except` 捕获，然后做一些异常处理：

```
try:
    if condition:
        revise_csv() # 已经污染csv文件

    do_something() # 触发异常
except Exception:
    handle_exception()
```

由于`try`块里的逻辑分为两步执行，它们不是一个原子操作，所以首先修改了csv文件，但是 `do_something` 却出现异常，导致污染csv文件。

其实，除了以上EAFP防御性编程风格外，还有一种编程风格与它截然不同，它虽然能很好的解决EAFP的副作用，但是缺点更加明显，所以Python中不太提倡大量的使用此种风格。

## 8 LBYL 防御编程风格

再介绍另一种编程风格：LBYL

它的特点：指在执行正常的业务逻辑前做好各种可能出错检查，需要写一个又一个的 `if` 和 `else` 逻辑。

如 EAFP 风格的代码：

```
d = {'a': 1, 'b': [1, 2, 3]}

try:
    val = d['c']
except KeyError:
    print('key not existence')
```

使用 LBYL 来写就是如下这样：

```
if 'c' in d:
    val = d['c']
else:
    print('key not existence')
```

EAFP 风格的代码如下：

```
import os
try:
    os.makedirs(path)
except OSError as exception:
    if exception.errno != errno.EEXIST:
        raise # PermissionError 等异常
    else:
        # path 目录已存在
```

使用 LBYL 来写就是如下这样：

```
import os

if not os.path.isdir(path):
    print('不是一个合法路径')

else:
    if not os.path.exists(path):
        os.makedirs(path)
    else:
        print('路径已存在')
```

通过以上两个例子，大家可以看出 LBYL 风格和 EAFP 风格迥异。

LBYL 的代码 if 和 else 较多，这种风格会有以下缺点。

## 9 程序每次运行都要检查

程序每次运行都要检查，不管程序是不是真的会触发这些异常。

```
if 'c' in d: # 每次必做检查
    val = d['c']

if not os.path.isdir(path): # 每次必做检查
    print('不是一个合法路径')

else:
    if not os.path.exists(path): # 每次必做检查
        os.makedirs(path)
    else:
        print('路径已存在')
```

## 10 很难一次考虑所有可能异常

很难一次性考虑到所有可能的异常，更让人头疼的事情是，一旦遗漏某些异常情况，错误经常不在出现的地方，而在很外层的一个调用处。这就会导致我们花很多时间调试才能找到最终出错的地方。

```
def f1()
    if con1:
        # do1()
    if con2:
        # do2()
# 但是遗漏了情况3，未在f1函数中报异常
```

## 11 代码的可读性下降

要写很多与主逻辑无关的 `if-else`，程序真正的逻辑就变得难以阅读。最后导致我们很难看出这个只是判断，还是程序逻辑/业务的判断。但是，如果用 `try-catch`，那么try代码块里面可以只写程序的逻辑，在 `except` 里面处理所有的异常。

结论：就Python语言，推荐使用 `EAFP` 风格，个别受保护的块，若无法实现原子操作的地方可以使用 `LBYL` 风格。

以上就是本专题编程风格的主要总结，原创不易，欢迎大家三连支持。

### Python 函数专题

函数是一个接受输入、进行特定计算并产生输出的语句集。

我们把一些经常或反复被使用的任务放在一起，创建一个函数，而不是为不同的输入反复编写相同的代码。

Python提供了 `print`、`sorted`、`max`、`map` 等内置函数，但我们也可以创建自己的函数，被称为用户定义函数。

## 六、Python函数专题

### 1 函数组成

如下自定义函数：

```
def foo(nums):
    """ 返回偶数序列 """
    evens = []
    for num in nums:
        if num%2==0:
            evens.append(num)
    return evens
```

可以看到函数主要组成部分：

- 函数名：`foo`
- 函数形参：`nums`
- `:`：函数体的控制字符，作用类似 `Java` 或 `C++` 的一对 `{}`
- 缩进：一般为4个字符
- `"""`：为函数添加注释
- `return`：函数返回值

以上函数求出列表 `nums` 中的所有偶数并返回，通过此函数了解Python函数的主要组成部分。

## 2 引用传参

定义好一个函数后，使用：函数名+()+实参，调用函数，如下方法：

```
foo([10,2,5,4])
```

其中 `[10,2,5,4]` 为实参，它通过 `by reference` 方式传给形参 `nums`，即 `nums` 指向列表头，而不是重新复制一个列表给 `nums`。

再看一个引用的例子：

```
def myFun(x):
    x[0] = 20
```

如下调用：

```
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
```

实参 `lst` 和形参 `x` 都指向同一个列表：



因此，对 `x[0]` 修改实际就是对实参 `lst` 的修改，结果如下：

但是，有时在函数内部形参指向改变，因此实参与形参的指向分离，如下例子：

```
def myFun(x):  
    x = [20, 30, 40]  
    x[0] = 0
```

调用：

```
lst = [10, 11, 12, 13, 14, 15]  
myFun(lst)
```

`x` 被传参后初始指向 `lst`，如下所示：

但是，执行 `x = [20, 30, 40]` 后，对象 `x` 重新指向一个新的列表对象 `[20,30,40]`：

因此，对于 `x` 内元素的任何修改，都不会同时影响到 `lst`，因为指向已经分离。

### 3 默认参数与关键字参数

Python函数的参数，可以有初始默认值，在调用时如果不赋值，则取值为默认值，如下例子：

```
def foo(length,width,height=1.0):  
    return length*width*height
```

调用 `foo` 函数，没有为 `height` 传参，所以取为默认值 `1.0`：

```
r = foo(1.2,2.0)  
print(r) # 2.4
```

使用默认值有一点需要区分，有的朋友会与关键字参数混淆，因为它们都是 `para=value` 的结构，但是有一个很明显的不同：默认值是声明在函数定义时，关键字参数是在函数调用时使用的此结构。如下例子：

```
def foo(length,width,height=1.0): # height是默认参数  
    return length*width*height
```

```
foo(width=2.0,length=1.2) #确定这种调用后才确定width和length是关键字参数
```

确定以上调用后，才确定 `width` 和 `length` 是关键字参数，并且关键字参数不必按照形参表的顺序调用。

## 4 可变参数

`Java` 和 `C++` 在解决同一个函数含有参数个数不同时，会使用函数重载的方法。`Python`使用可变参数的方法，并且非常灵活。

可变参数是指形参前带有 `*` 的变量，如下所示：

```
def foo(length, *others):  
    s = length  
    for para in others:  
        s *= para  
    return s
```

我们可以像下面这样方便的调用：

```
foo(1.2, 2.0, 1.0) # 2.4
```

如上，带一个星号的参数被传参后，实际被解释为 `元组对象`。我们还可以这样调用：

```
foo(1.2) # 1.2
```

## 5 内置函数

总结完函数的参数后，再举几个`Python`内置的常用函数。

### `pow`

大部分朋友应该知道 `pow` 是个幂次函数，比如求

$2^3$   
：

```
pow(2, 3)
```

除此以外，`pow` 还有第三个参数，使用更高效的算法实现求幂后再求余数操作，

```
pow(2,3,5) # 3
```

## max,min

max,min用来求解最大最小值，实现 `relu` 函数：

```
def relu(x):  
    return max(x,0)
```

## sorted

sorted函数完成对象排序，它能接收一个指定排序规则的函数，完成定制排序。如下，根据字典值绝对值从小到大排序：

```
d = {'a':0, 'b':-2, 'c':1}  
dr = sorted(d.items(),key=lambda x:abs(x[1]))  
print(dr) # [('a', 0), ('c', 1), ('b', -2)]
```

Python有一个专门操作函数的模块：`functools`，能实现一些关于函数的特殊操作。

## 6 偏函数

偏函数固定函数的某些参数后，重新生成一个新的函数。

通常用法，当函数的参数个数太多，需要简化时，使用 `partial` 可以创建一个新的函数。

假设我们要经常调用 `int` 函数转换二进制字符，设置参数 `base` 为2:

```
int('1010',base=2)
```

为了免去每次都写一个参数`base`，我们重新定义一个函数：

```
def int2(s):  
    return int(s,base=2)
```

以后每次转化字符串时，只需 `int2('1010')` 即可，更加简便。

偏函数也能实现上述功能：

```
from functools import partial

intp = partial(int, base=2)
```

那么有的朋友会问，偏函数就是个鸡肋，重新定义的 `int2` 更加直观容易理解，这个角度讲确实是这样。但是 `int2` 不能再接收 `base` 参数，但是 `intp` 函数还是能接收 `base` 参数，依然保留了原来的参数：

```
intp('10', base=16) # 16
```

可能看到这里的读者还是有些迷糊，不太确定怎么使用偏函数。可以先记住：修改内置函数的默认参数，就像内置函数 `int` 默认参数 `base` 等于10，使用偏函数调整为2.

## 7 递归函数

递归函数是指调用自身的函数。如使用递归反转字符串：

```
def reverseStr(s):
    if not s:
        return s
    return reverseStr(s[1:])+s[0]

print(reverseStr('nohtyp')) # python
```

`reverseStr` 函数里面又调用了函数 `reverseStr`，所以它是递归函数。

使用递归函数需要注意找到正确的递归基，防止陷入无限递归。

更多使用递归的例子大家可参考此公众号之前推送。

## 8 匿名函数

匿名函数是指使用 `lambda` 关键字创建的函数。它的标准结构如下：

```
lambda 形参列表: 含有形参列表的表达式
```

表达式的计算值即为 `lambda` 函数的返回值。

如下 `lambda` 函数：

```
lambda x,y: x+y
```

它等价于下面的 `f` 函数:

```
def f(x,y):  
    return x+y
```

lambda函数常做为 `max`, `sorted`, `map`, `filter` 等函数的key参数。

## 9 高阶函数

可以用来接收另一个函数作为参数的函数叫做高阶函数。

如下 `f` 有一个参数 `g`，而 `g` 又是函数，所以 `f` 是高阶函数：

```
def f(g):  
    g()
```

Python 中经常会遇到高阶函数，今天介绍几个内置的常用的高阶函数。

### map

`map` 函数第一个参数为函数，它作用于列表中每个的元素。

如下，列表中的单词未按照首字母大写其他字符小写的规则，使用 `map` 一一 `capitalize` 每个元素：

```
m = map(lambda s: s.capitalize(), ['python', 'Very', 'BEAUTIFUL'])  
print(list(m))
```

结果：

```
['Python', 'Very', 'Beautiful']
```

### reduce

`reduce` 高阶函数实现化简列表，它实现的效果如下：

$$reduce(x1, x2, x3) = f(f(x1, x2), x3)$$

如下例子，函数 `f` 等于 `x+y`，求得两数之和，然后再与第三个数相加，依次下去，直到列表尾部，进而得到整个列表的和：

```
from functools import reduce

def f(x,y):
    return x+y

r = reduce(f, [1,3,2,4])
print(r) # 10
```

以上 `reduce` 求解过程等于：

$$\begin{aligned} & f(f(f(1,3),2),4) \\ &= f(f(4,2),4) \\ &= f(6,4) \\ &= 10 \end{aligned}$$

需要注意：`reduce` 函数要求 `f` 必须带2个参数，只有这样才能完成归约化简。

## 10 嵌套函数

嵌套函数是指里面再嵌套函数的函数。

如下例子，将列表转化为二叉树。已知列表 `nums`，

`nums = [3,9,20,None,None,15,7]`，转化为下面二叉树：

二叉树定义：

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

构建满足以上结构的二叉树，可以观察到：树的父节点和左右子节点的关系：

$$\begin{aligned} left_{index} &= 2 \times parent_{index} + 1 \\ right_{index} &= 2 \times parent_{index} + 2 \end{aligned}$$

基于以上公式，再使用递归构建二叉树。

递归基情况：

```
if index >= len(nums) or nums[index] is None:
    return None
```

递归方程：

$$left_{node} = f(2 \times parent_{index} + 1)$$

$$right_{node} = f(2 \times parent_{index} + 2)$$

根据以上分析，得到如下代码，`list_to_binarytree` 函数是嵌套函数，它里面还有一个 `level` 子函数：

```
def list_to_binarytree(nums):
    def level(index):
        if index >= len(nums) or nums[index] is None:
            return None

        root = TreeNode(nums[index])
        root.left = level(2 * index + 1)
        root.right = level(2 * index + 2)
        return root

    return level(0)

binary_tree = list_to_binarytree([3, 9, 20, None, None, 15, 7])
```

通常使用嵌套函数的场景：实现一个功能只需要编写2个函数，写成一个 `class` 好像显得有些不必要，写成嵌套后某些参数能共享，亲和性会更好。不妨体会上面的 `nums` 参数。

## 七、Python 面向对象编程(上篇)

面向对象程序设计思想，首先思考的不是程序执行流程，它的核心是抽象出一个对象，然后构思此对象包括的数据，以及操作数据的行为方法。

### 1 类定义

动物是自然界一个庞大的群体，以建模动物类为主要案例论述OOP编程。

Python语言创建动物类的基本语法如下，使用 `class` 关键字定义一个动物类：

```
class Animal():  
    pass
```

类里面可包括数据，如下所示的 `Animal` 类包括两个数据：`self.name` 和 `self.speed`：

```
class Animal():  
    def __init__(self, name, speed):  
        self.name = name # 动物名字  
        self.speed = speed # 动物行走或飞行速度
```

注意到类里面通过系统函数 `__init__` 为类的2个数据赋值，数据前使用 `self` 保留字。

`self` 的作用是指名这两个数据是实例上的，而非类上的。

同时注意到 `__init__` 方法的第一个参数也带有 `self`，所以也表明此方法是实例上的方法。

## 2 实例

理解什么是实例上的数据或方法，什么是类上的数据，需要先建立 `实例` 的概念，`类` 的概念，如下：

```
# 生成一个名字叫加菲猫、行走速度8km/h的cat对象  
cat = Animal('加菲猫', 8)
```

`cat` 就是 `Animal` 的实例，也可以一次创建成千上百个实例，如下创建1000只蜜蜂：

```
bees = [Animal('bee'+str(i), 5) for i in range(1000)]
```

总结：自始至终只使用一个类 `Animal`，但却可以创建出许多个它的实例，因此是一对多的关系。

实例创建完成后，下一步打印它看看：

```
In [1]: print(cat)  
<__main__.Animal object at 0x7fce3a596ad0>
```

结果显示它是 `Animal` 对象，其实打印结果显示实例属性信息会更友好，那么怎么实现呢？



## 3 打印实例

只需重新定义一个系统(又称为魔法)函数 `__str__`，就能让打印实例显示的更加友好：

```
class Animal():
    def __init__(self, name, speed):
        self.name = name # 动物名字
        self.speed = speed # 动物行走或飞行速度

    def __str__(self):
        return '''Animal({0.name},{0.speed}) is printed
                name={0.name}
                speed={0.speed}'''.format(self)
```

使用 `{0.数据名称}` 的格式，这是类专有的打印格式。

现在再打印：

```
cat = Animal('加菲猫',8)
print(cat)
```

打印信息如下：

```
Animal(加菲猫,8) is printed
        name=加菲猫
        speed=8
```

以上就是想要的打印格式，看到实例的数据值都正确。

## 4 属性

至此，我们都称类里的 `name` 和 `speed` 为数据，其实它们有一个专业名称：属性。

同时，上面还有一个问题我们没有回答完全，至此已经解释了实例的属性。那么什么是类上的属性？

如下，在最新 `Animal` 类定义基础上，再添加一个 `cprop` 属性，它前面没有 `self` 保留字：

```
class Animal():
    cprop = "我是类上的属性cprop"

    def __init__(self,name,speed):
        self.name = name # 动物名字
        self.speed = speed # 动物行走或飞行速度

    def __str__(self):
        return '''Animal({0.name},{0.speed}) is printed
        name={0.name}
        speed={0.speed}'''.format(self)
```

类上的属性直接使用类便可引用：

```
In [1]: Animal.cprop
Out[1]: '我是类上的属性cprop'
```

类上的属性，实例同样可以引用，并且所有的实例都共用此属性值：

```
In [1]: cat = Animal('加菲猫',8)
In [2]: cat.cprop
Out[2]: '我是类上的属性cprop'
```

Python作为一门动态语言，支持属性的动态添加和删除。

如下 `cat` 实例原来不存在 `color` 属性，但是赋值时不光不会报错，相反会直接将属性添加到 `cat` 上：

```
cat.color = 'grap'
```

那么，如何验证 `cat` 是否有 `color` 属性呢？使用内置函数 `hasattr`：

```
In [24]: hasattr(cat,'color') # cat 已经有`color`属性
Out[24]: True
```

但是注意：以上添加属性方法仅仅为 `cat` 实例本身添加，而不会为其他实例添加：

```
In [26]: monkey = Animal('大猩猩',2)
In [27]: hasattr(monkey,'color')
Out[27]: False
```

`monkey` 实例并没有 `color` 属性，注意与 `__init__` 创建属性方法的区别。

## 5 private,protected,public

像 `name` 和 `speed` 属性，引用此实例的对象都能访问到它们，如下：

```
# 模块名称: manager.py

import time

class Manager():
    def __init__(self, animal):
        self.animal = animal

    def recordTime(self):
        self.__t = time.time()
        print('feeding time for %s (行走速度为:%s) is %.0f'%(self.animal.name, self.animal.speed, self.__t))

    def getFeedingTime(self):
        return '%0.1f'%(self.__t,)
```

使用以上 `Manager` 类，创建一个 `cat` 实例，`xiaoming` 实例引用 `cat`：

```
cat = Animal('加菲猫', 8)
xiaoming = Manager(cat)
```

`xiaoming` 的 `recordTime` 方法引用里，引用了 `animal` 的两个属性 `name` 和 `speed`：

```
In[1]: xiaoming.recordTime()

Out[1]: feeding time for 加菲猫 (行走速度为:8) is 1595681304
```

注意看到 `self.__t` 属性，它就是一个私有属性，只能被 `Manager` 类内的所有方法引用，如被方法 `getFeedingTime` 方法引用。但是，不能被其他类引用。

如果我们连 `speed` 这个属性也不想被其他类访问，那么只需稍作下 `Animal` 类，将 `self.speed` 修改为 `self.__speed`：

同时修改 `Manager` 类的 `self.animal.speed` 修改为 `self.animal.__speed`，再次调用下面方法时：

```
xiaoming.recordTime()
```

就会报没有 `__speed` 属性的异常，从而验证了 `__speed` 属性已经变为类内私有，不会暴露在外面。

总结：`name` 属性相当于java的public属性，而 `__speed` 相当于java的private属性。

下面在说继承时，讲解 `protected` 属性，实际上它就是带有1个 `_` 的属性，它只能被继承的类所引用。

## 6 继承

上面已经讲完了OOP三大特性中的封装性，而继承是它的第二大特性。子类继承父类的所有 `public` 和 `protected` 数据和方法，极大提高了代码的重用性。

如上我们创建的 `Animal` 类最新版本为：

```
class Animal():
    cprop = "我是类上的属性cprop"

    def __init__(self, name, speed):
        self.name = name # 动物名字
        self.__speed = speed # 动物行走或飞行速度

    def __str__(self):
        return '''Animal({0.name},{0.__speed}) is printed
                name={0.name}
                speed={0.__speed}'''.format(self)
```

现在有个新的需求，要重新定义一个 `Cat` 猫类，它也有 `name` 和 `speed` 两个属性，同时还有 `color` 和 `genre` 两个属性，打印时只需要打印 `name` 和 `speed` 两个属性就行。

因此，基本可以复用基类 `Animal`，但需要修改 `__speed` 属性为受保护( `protected` )的 `_speed` 属性，这样子类都可以使用此属性，而外部还是访问不到它。

综合以上，`Cat` 类的定义如下：

```
class Cat(Animal):
    def __init__(self, name, speed, color, genre):
        super().__init__(name, speed)
        self.color = color
        self.genre = genre
```

首先使用 `super()` 方法找到 `Cat` 的基类 `Animal`，然后引用基类的 `__init__` 方法，这样复用了基类的此方法。

使用 `Cat` 类，打印时，又复用了基类的 `__str__` 方法：

```
jiafeimao = Cat('加菲猫', 8, 'gray', 'CatGenre')
print(jiafeimao)
```

打印结果：

```
Animal(加菲猫, 8) is printed
      name=加菲猫
      speed=8
```

以上就是基本的继承使用案例，继承要求基类定义的数据和行为尽量标准、尽量精简，以此提高继承的复用性。

## 7 多态

如果说OOP的封装和继承使用起来更加直观易用，那么作为第三大特性的多态，在实践中真正运用起来可就不那么容易。有的读者OOP编程初期，可能对多态的价值体会不深刻，甚至都已经淡忘它的存在。

那么问题就在：多态到底真的有用吗？到底使用在哪些场景？

多态价值很大，使用场景很多，几乎所有的系统或软件，都能看到它的应用。

这篇文章，我会尽可能通过一个精简的例子说明它的价值和使用方法。

使用对比法，如果不用多态，方法怎么写；使用多态，又是怎么写。

为了一脉相承，做到一致性，仍然基于上面的案例，已经创建好的 `Cat` 类要有一个方法打印和返回它的爬行速度。同时需要再创建一个类 `Bird`，要有一个方法打印和返回它的飞行速度；

如果不使用多态，可能会这样写：

对于 `Cat` 类只需新增一个方法：

```
class Cat(Animal):
    def __init__(self, name, speed, color, genre):
        super().__init__(name, speed)
        self.color = color
        self.genre = genre
    # 添加方法
    def getRunningSpeed(self):
        print('running speed of %s is %s' %(self.name, self._speed))
        return self._speed
```

重新创建一个 `Bird` 类：

```
class Bird(Animal):
    def __init__(self, name, speed, color, genre):
        super().__init__(name, speed)
        self.color = color
        self.genre = genre
    # 添加方法
    def getFlyingSpeed(self):
        print('flying speed of %s is %s' %(self.name, self._speed))
        return self._speed
```

最后，上面创建的 `Manager` 类会引用 `Cat` 和 `Bird` 类，但是需要修改 `recordTime` 方法，因为一个为 `Cat` 它是爬行的，`Bird` 它是飞行的，所以要根据类型不同做逻辑区分，如下所示：

```

# 模块名称: manager.py

import time
from animal import (Animal, Cat, Bird)

class Manager():
    def __init__(self, animal):
        self.animal = animal

    def recordTime(self):
        self.__t = time.time()
        if isinstance(self.animal, Cat):
            print('feeding time for %s is %.0f'%(self.animal.name, self.__t))
            self.animal.getRunningSpeed()
        if isinstance(self.animal, Bird):
            print('feeding time for %s is %.0f'%(self.animal.name, self.__t))
            self.animal.getFlyingSpeed()

    def getFeedingTime(self):
        return '%0.0f'%(self.__t,)

```

如果再来一个类，我们又得需要修改 `recordTime`，再增加一个 `if` 分支，从软件设计角度讲，这种不断破坏原来类方法的行为不可取。

但是，使用多态，就可以保证 `recordTime` 不被修改，不必写很多 `if` 分支。

怎么来实现呢？一种实现方法：首先在基类 `Animal` 中创建一个基类方法，然后 `Cat` 和 `Bird` 分别重写此方法，最后传入到 `Manager` 类的 `animal` 参数是什么类型，在 `recordTime` 方法中就会对应调用这个 `animal` 实例的方法，这就是多态。

代码如下：

[animal2.py](#) 模块如下：

# animal2.py 模块

```
class Animal():
    cprop = "我是类上的属性cprop"

    def __init__(self, name, speed):
        self.name = name # 动物名字
        self._speed = speed # 动物行走或飞行速度

    def __str__(self):
        return '''Animal({0.name},{0._speed}) is printed
                name={0.name}
                speed={0._speed}'''.format(self)

    def getSpeedBehavior(self):
        pass

class Cat(Animal):
    def __init__(self, name, speed, color, genre):
        super().__init__(name, speed)
        self.color = color
        self.genre = genre

    # 添加方法
    def getSpeedBehavior(self):
        print('running speed of %s is %s' %(self.name, self._speed))
        return self._speed

class Bird(Animal):
    def __init__(self, name, speed, color, genre):
        super().__init__(name, speed)
        self.color = color
        self.genre = genre

    # 添加方法
    def getSpeedBehavior(self):
        print('flying speed of %s is %s' %(self.name, self._speed))
        return self._speed
```

manager2.py 模块如下：



```
# manager2.py 模块

import time
from animal2 import (Animal, Cat, Bird)

class Manager():
    def __init__(self, animal):
        self.animal = animal

    def recordTime(self):
        self.__t = time.time()
        print('feeding time for %s is %.0f'%(self.animal.name, self.__t))
        self.animal.getSpeedBehavior()

    def getFeedingTime(self):
        return '%0.f'%(self.__t,)
```

`recordTime` 方法非常清爽，不需要任何if逻辑，只需要调用我们定义的 `Animal` 类的基方法 `getSpeedBehavior` 即可。

在使用上面所有类时，`Manager(jiafeimao)` 传入 `Cat` 类实例时，`recordTime` 方法调用就被自动指向 `Cat` 实例的 `getSpeedBehavior` 方法；

`Manager(haiying)` 传入 `Bird` 类实例时，自动指向 `Bird` 实例的 `getSpeedBehavior` 方法，这就是多态和它的价值。

```
if __name__ == "__main__":
    jiafeimao = Cat('jiafeimao', 2, 'gray', 'CatGenre')
    haiying = Bird('haiying', 40, 'blue', 'BirdGenre')

    Manager(jiafeimao).recordTime()
    print('#'*30)
    Manager(haiying).recordTime()
```

## 八、Python面向对象编程(下篇)

### 1 创建抽象方法

上篇讲解多态部分，定义了基类模块 `animals2.py`，它里面有一个方法 `getSpeedBehavior`，然后2个继承类中分别重写了此方法。虽然这种模式并不会报错，但却不是最佳编程写法。

```

class Animal():
    cprop = "我是类上的属性cprop"

    def __init__(self, name, speed):
        self.name = name # 动物名字
        self._speed = speed # 动物行走或飞行速度

    def __str__(self):
        return '''Animal({0.name},{0._speed}) is printed
                name={0.name}
                speed={0._speed}'''.format(self)

    def getSpeedBehavior(self):
        pass

```

更加优秀的做法，显示的定义基类的此方法为抽象方法，并且明确指名这两个继承类需要重写此方法。

借助Python内置的 `abc` 模块，使用 `abstractmethod` 装饰器，`Animal` 类的改进版：

```

import abc

class Animal():
    cprop = "我是类上的属性cprop"

    def __init__(self, name, speed):
        self.name = name # 动物名字
        self._speed = speed # 动物行走或飞行速度

    def __str__(self):
        return '''Animal({0.name},{0._speed}) is printed
                name={0.name}
                speed={0._speed}'''.format(self)

    # 使用abstractmethod装饰器后，变为抽象方法
    @abc.abstractmethod
    def getSpeedBehavior(self):
        pass

```

其他类都不改变。以上就是创建抽象类的方法。

## 2 检查属性取值

已经在Animal类中定义2个属性name和\_speed：

```
class Animal():
    cprop = "我是类上的属性cprop"

    def __init__(self, name, speed):
        self.name = name # 动物名字
        self._speed = speed # 动物行走或飞行速度
```

像这种方法定义的属性，外界可以对属性赋任意值，这不是合理的。如下speed参数被赋值为负值，这肯定不合理：

```
jiafeimao = Cat('jiafeimao', -2, 'gray', 'CatGenre')
```

所以一种解决方法便是使用@property，写法也很简洁：

```
# 读
@property
def _speed(self):
    return self.__speed
# 写
@_speed.setter
def _speed(self, val):
    if val < 0:
        raise ValueError('speed value is negative')
    self.__speed = val
```

Cat('jiafeimao', -2, 'gray', 'CatGenre')执行时，会进入到@\_speed.setter，检查不满足，抛出取值异常。

@property就是给\_speed函数增加功能后返回一个更强大的函数，@属性.setter也是一个函数，装饰后控制着属性的写入操作。

## 3 给类添加属性

基础篇说到为实例添加属性，只对此实例生效，其他属性还是没有此属性。怎样在外面一次添加属性后，所有实例都能具有呢。

答案是为类添加属性，如下所示，为Cat类增加属性age后，jiafeimao实例和jiqimao实例都有了age属性，且都可被修改：

```
if __name__ == "__main__":
    jiafeimao = Cat('jiafeimao', 2, 'gray', 'CatGenre')

    Cat.age = 1
    jiafeimao.age = 3
    print(jiafeimao.age) # 3
    jiqimao = Cat('jiqimao', 3, 'dark', 'CatGenre')
    jiqimao.age = 5
    print(jiqimao.age) # 5
```

这就说明，一次为类添加一个属性，类的所有实例都会有这个新增的属性。

这种虽然写法便利，但是会带来副作用，支持动态添加实际上破坏了类的封装性，为维护程序带来不便。同时，如果泛滥使用，属性过多占用内存就会变大，影响程序的性能。

## 4 控制随意添加属性

Python应该意识到上面动态添加属性带来的副作用，因此留出一个系统魔法函数 `__slots__`，以此来控制随意在外添加属性。

使用 `__slots__`，定义这个类只能有哪些属性，不在这个元组里的属性添加都会失败。

如下这样做后，控制Student类只能有属性name和age，不允许添加其他属性：

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称

    def __init__(self, name, age):
        self.name = name
        self.age = age

s = Student('xiaoming', 100) # 创建新的实例
s.score = 10
```

如下异常：

## Exception has occurred: AttributeError

'Student' object has no attribute 'score'

File `"/home/zglg/mywork/mdfiles/test.py"`, line 10, in `<module>`  
`s.score=10`

## 5 链式调用

每个对外公开的方法，都返回`self`，这样在外面调用时，便能形成一条链式调用线，在`pyecharts`等框架中可以看到这种调用风格。

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称

    def __init__(self, name, age):
        self.name = name
        self.age = age

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

    def set_age(self, age):
        self.age = age
        return self

    def print_info(self):
        print("name: "+self.name)
        print("age: "+str(self.age))
        return self

s = Student('xiaoming', 100) # 创建新的实例

(
    s
    .set_name('xiaoming1')
    .set_age(25)
    .print_info()
)
```

关于面向对象编程的进阶部分，还有一个重要的设计原则：`Mixin` 原则，这个我们放到后面在讲设计模式时一起讨论。

# 九、Python十大数据结构

## 简介

这个专题，尽量使用最精简的文字，借助典型案例盘点Python常用的数据结构。

如果你还处于Python入门阶段，通常只需掌握 `list`、`tuple`、`set`、`dict` 这类数据结构，做到灵活使用即可。

然而，随着学习的深入，平时遇到实际场景变复杂，很有必要去了解Python内置的更加强大的数据结构 `deque`、`heapq`、`Counter`、`OrderedDict`、`defaultDict`、`ChainMap`，掌握它们，往往能让你少写一些代码且能更加高效的实现功能。

学习数据结构第一阶段：掌握它们的基本用法，使用它们解决一些基本问题；

学习第二阶段：知道何种场景选用哪种最恰当的数据结构，去解决问题；

学习第三阶段：了解内置数据结构的背后源码实现，与《算法和数据结构》这门学问里的知识联系起来，打通任督二脉。

下面根据定义的这三个阶段，总结以下**10**种最常用的数据结构：

## 1 list

**基本用法** 废话不多说，在前面单独有一个专题详述了list的使用【[添加文章链接](#)】

**使用场景** list 使用在需要查询、修改的场景，极不擅长需要频繁插入、删除元素的场景。

**实现原理** list对应数据结构的线性表，列表长度在初始状态时无需指定，当插入元素超过初始长度后再启动动态扩容，删除时尤其位于列表开始处元素，时间复杂度为 $O(n)$

## 2 tuple

元组是一类不允许添加删除元素的特殊列表，也就是一旦创建后续决不允许增加、删除、修改。

**基本用法** 元组大量使用在打包和解包处，如函数有多个返回值时打包为一个元组，赋值到等号左侧变量时解包。

```
In [22]: t=1,2,3
In [23]: type(t)
Out[23]: tuple
```

实际创建一个元组实例

**使用场景** 如果非常确定你的对象后面不会被修改，则可以大胆使用元组。为什么？因为相比于list, tuple实例更加节省内存，这点尤其重要。

```
In [24]: from sys import getsizeof

In [25]: getsizeof(list())
Out[25]: 72 # 一个list实例占用72个字节

In [26]: getsizeof(tuple())
Out[26]: 56 # 一个tuple实例占用56个字节
```

所以创建100个实例，tuple能节省1K多字节。

### 3 set

**基本用法** set是一种里面不能含有重复元素的数据结构，这种特性天然的使用于列表的去重。

```
In [27]: a=[3,2,5,2,5,3]

In [28]: set(a)
Out[28]: {2, 3, 5}
```

除此之外，还有知道set结构可用于两个set实例的求交集、并集、差集等操作。

```
In [29]: a = {2,3,5}

In [30]: b = {3,4,6,2}

In [31]: a.intersection(b) # 求交集
Out[31]: {2, 3}
```

**使用场景** 如果只是想缓存某些元素值，且要求元素值不能重复时，适合选用此结构。并且set内允许增删元素，且效率很高。

**实现原理** set在内部将值哈希为索引，然后按照索引去获取数据，因此删除、增加、查询元素效

果都很高。

## 4 dict

**基本用法** dict 是Python中使用最频繁的数据结构之一，字典创建由通过dict函数、{}写法、字典生成式等，增删查元素效率都很高。

```
d = {'a':1, 'b':2} # {}创建字典

# 列表生成式
In [38]: d = {a:b for a,b in zip(['a', 'b'], [1,2])}
In [39]: d
Out[39]: {'a': 1, 'b': 2}
```

**使用场景** 字典尤其适合在查询多的场景，时间复杂度为 $O(1)$ 。如leetcode第一题求解两数之和时，就会使用到dict的 $O(1)$ 查询时间复杂度。

同时，Python类中属性值等信息也都是缓存在 `__dict__` 这个字典型数据结构中。

但是值得注意，dict占用字节数是list、tuple的3、4倍，因此对内存要求苛刻的场景要慎重考虑。

```
In [40]: getsizeof(dict())
Out[40]: 248
```

**实现原理** 字典是一种哈希表，同时保存了键值对。

以上4种数据结构相信大家都已经比较熟悉，因此我言简意赅的介绍一遍。接下来再详细的介绍下面6种数据结构及各自使用场景，会列举更多的例子。

## 5 deque

**基本用法** deque 双端队列，基于list优化了列表两端的增删数据操作。基本用法：



```
from collections import deque

In [3]: d = deque([3,2,4,0])

In [4]: d.popleft() # 左侧移除元素, O(1)时间复杂度
Out[4]: 3

In [5]: d.appendleft(3) # 左侧添加元素, O(1)时间复杂度

In [6]: d
Out[6]: deque([3, 2, 4, 0])
```

使用场景 list左侧添加删除元素的时间复杂度都为 $O(n)$ ，所以在Python模拟队列时切忌使用list，相反使用deque双端队列非常适合频繁在列表两端操作的场景。但是，加强版的deque牺牲了空间复杂度，所以嵌套deque就要仔细trade-off:

```
In [9]: sys.getsizeof(deque())
Out[9]: 640

In [10]: sys.getsizeof(list())
Out[10]: 72
```

实现原理 cpython实现deque使用默认长度64的数组，每次从左侧移除1个元素，leftindex加1，如果超过64释放原来的内存块，再重新申请64长度的数组，并使用双端链表block管理内存块。

## 6 Counter

基本用法 Counter一种继承于dict用于统计元素个数的数据结构，也称为bag 或 multiset. 基本用法：

```
from collections import Counter

In [14]: c = Counter([1,3,2,3,4,2,2]) # 统计每个元素的出现次数
In [17]: c
Out[17]: Counter({1: 1, 3: 2, 2: 3, 4: 1})

# 除此之外，还可以统计最常见的项
# 如统计第1最常见的项，返回元素及其次数的元组
In [16]: c.most_common(1)
Out[16]: [(2, 3)]
```

使用场景 基本的dict能解决的问题就不要用Counter，但如遇到统计元素出现频次的场景，就不

要自己去用dict实现了，果断选用Counter.

需要注意，Counter统计的元素要求可哈希(hashable)，换句话说如果统计list的出现次数就不可行，不过list转化为tuple不就可哈希了吗.

实现原理 Counter实现基于dict，它将元素存储于keys上，出现次数为values.

## 7 OrderedDict

基本用法 继承于dict，能确保keys值按照顺序取出来的数据结构，基本用法：

```
In [25]: from collections import OrderedDict

In [26]: od = OrderedDict({'c':3, 'a':1, 'b':2})

In [27]: for k,v in od.items():
...:     print(k,v)
...:
c 3
a 1
b 2
```

使用场景 基本的dict无法保证顺序，keys映射为哈希值，而此值不是按照顺序存储在散列表中的。所以遇到要确保字典keys有序场景，就要使用OrderedDict.

实现原理 你一定会好奇OrderedDict如何确保keys顺序的，翻看cpython看到它里面维护着一个双向链表 `self.__root`，它维护着keys的顺序。既然使用双向链表，细心的读者可能会有疑问：删除键值对如何保证O(1)时间完成？

cpython使用空间换取时间的做法，内部维护一个 `self.__map` 字典，键为key，值为指向双向链表节点的 `link`。这样在删除某个键值对时，通过\_\_map在O(1)内找到link，然后O(1)内从双向链表\_\_root中摘除。

## 8 heapq

基本用法 基于list优化的一个数据结构：堆队列，也称为优先队列。堆队列特点在于最小的元素总是在根结点：`heap[0]` 基本用法：

```

import heapq
In [41]: a = [3,1,4,5,2,1]

In [42]: heapq.heapify(a) # 对a建堆，建堆后完成对a的就地排序
In [43]: a[0] # a[0]一定是最小元素
In [44]: a
Out[44]: [1, 1, 3, 5, 2, 4]

In [46]: heapq.nlargest(3,a) # a的前3个最大元素
Out[46]: [5, 4, 3]

In [47]: heapq.nsmallest(3,a) # a的前3个最小元素
Out[47]: [1, 1, 2]

```

**使用场景** 如果想要统计list中前几个最小(大)元素，那么使用heapq很方便，同时它还提供合并多个有序小list为大list的功能。

**基本原理** 堆是一个二叉树，它的每个父节点的值都只会小于或大于所有孩子节点（的值），原理与堆排序极为相似。

## 9 defaultdict

**基本用法** defaultdict是一种带有默认工厂的dict，如果对设计模式不很了解的读者可能会很疑惑工厂这个词，准确来说工厂全称为对象工厂。下面体会它的基本用法。

基本dict键的值没有一个默认数据类型，如果值为list，必须要手动创建：

```

words=['book','nice','great','book']
d = {}
for i,word in enumerate(words):
    if word in d:
        d[word].append(i)
    else:
        d[word]=[i] # 显示的创建一个list

```

但是使用defaultdict：

```

from collections import defaultdict
d = defaultdict(list) # 创建字典值默认为list的字典
for i,word in enumerate(words):
    d[word] = i

```

省去一层if逻辑判断，代码更加清晰。上面defaultdict(list)这行代码默认创建值为list的字典，还可以构造defaultdict(set), defaultdict(dict)等等，这种模式就是对象工厂，工厂里能制造各种对象：list,set,dict...

**使用场景** 上面已经说的很清楚，适用于键的值必须指定一个默认值的场景，如键的值为list,set,dict等。

**实现原理** 基本原理就是调用工厂函数去提供缺失的键的值。后面设计模式专题再详细探讨。

## 10 ChainMap

**基本用法** 如果有多个dict想要合并为一个大dict，那么ChainMap将是你的选择，它的方便性体现在同步更改。具体来看例子：

```
In [55]: from collections import ChainMap

In [56]: d1 = {'a':1, 'c':3, 'b':2}

In [57]: d2 = {'d':1, 'e':5}

In [58]: dm = ChainMap(d1,d2)

In [59]: dm
Out[59]: ChainMap({'a': 1, 'c': 3, 'b': 2}, {'d': 1, 'e': 5})
```

ChainMap后返回一个大dict视图，如果修改其对应键值对，原小dict也会改变：

```
In [86]: dm.maps # 返回一个字典list
Out[86]: [{'a': 2, 'c': 3, 'b': 2, 'd': 10}, {'d': 1, 'e': 5}]

In [87]: dm.maps[0]['d']=20 # 修改第一个dict的键等于'd'的值为20

In [88]: dm
Out[88]: ChainMap({'a': 2, 'c': 3, 'b': 2, 'd': 20}, {'d': 1, 'e': 5})

In [89]: d1 # 原小dict的键值变为20
Out[89]: {'a': 2, 'c': 3, 'b': 2, 'd': 20}
```

**使用场景** 具体使用场景是我们有多个字典或者映射，想把它们合并成为一个单独的映射，有读者可能说可以用update进行合并，这样做的问题就是新建了一个内存结构，除了浪费空间外，

还有一个缺点就是我们对新字典的更改不会同步到原字典上。

**实现原理** 通过maps便能观察出ChainMap联合多个小dict装入list中，实际确实也是这样实现的，内部维护一个lis实例，其元素为小dict.

## 十、Python 包和模块使用注意事项

今天这个专题讨论Python代码工程化、结构化的方法。我们都会遇到这种情景：所有代码都堆积到一个模块里，导致代码越来越长，最后变得难以维护，很明显代码只写到一个py模块文件是不可取的。如何按照逻辑功能，将代码划分到不同模块，组织为一个更易读、更易维护的代码结构呢？欢迎学习这个专题。

### 1 包和模块的定义

包(package)是一个文件夹，它里面会有一个 `__init__.py`，还有我们自己定义的.py文件。

而我们自己定义的.py文件，python中称为模块(module)，一个模块就是一个py文件，里面封装了一个功能模块，可能有函数、类、变量等。

如下建立的一个代码结构：

```
classdemo/
├── animals
│   ├── animal2.py
│   ├── animal.py
│   ├── __init__.py
│   ├── manager2.py
│   └── manager.py
└── search
    ├── binarytree_level.py
    └── __init__.py
```

里面包括两个package,一个为animals包，另一个为search包. 每个package里都有一个 `__init__.py` 文件。

使用这种结构带来什么便利？每个模块间的变量又该如何引用？里面的 `__init__.py` 起到什么作用？下面一一解答。

## 2 解决变量命名冲突

对程序员而言，变量命名往往是一个很头疼的难题，并且一不小心就会写出名称相同的变量，尤其是在同一模块里变量名称重复会很麻烦。

通常来说，一个模块里定义的代码行数不要太多，尽量拆分到不同的模块里，不同的模块允许出现相同名称的变量，这是划分不同模块的作用之一。

但是仅有模块好像还不够，对于一个大点的框架，再按照大的功能逻辑划分出包(package)显得更加有必要。

并且有了package后，相同变量名字冲突的可能性会更小。如第1小节中的 `Animal` 类，它的完整名称实际为：`animals.animal2.Animal`，这样在使用`Animal`等类时，导入方法是下面这样：

```
from animals.animal2 import (Animal2,Cat,Bird)
```

实际上，这种层级的组织在一些大的框架中到处可见。

## 3 `__init__.py` 作用

如上所述，`__init__.py` 会使得普通的文件夹变为package. 实际上，`__init__.py` 也是一个模块，其名称正是package的名字。

一般来说此文件为空，如下导入 `animals` 包：

```
In [2]: import animals
In [3]: animals
Out[3]: <module 'animals' from '/home/zglg/mywork/mdfiles/classdemo/animals/__init__.py'
```

可以看到导入 `animals` 包实际上导入了它下面的 `__init__.py` 文件。

同时还可以为它增加其他功能。

因为在导入一个包时实际上导入它的 `__init__.py` 文件，利用此特性，可以在 `__init__.py` 文件中批量导入多个模块都在公用的模块，而不再需要一个一个的导入。

拿上面的demo来说，`manager.py`和`manager2.py`中都用到 `time` 模块，我们就其移动到 `__init__.py` 里：

```
# __init__.py

import time
import os
import sys
import abc
```

在使用这些内置等模块时，首先导入包：

```
import animals # 导入包
```

在调用time模块时，必须使用**包名+模块名**的方式引用：

```
def recordTime(self):
    #引用变为: 包名animals + 模块名称
    self.__t = animals.time.time()
    print('feeding time for %s is %.0f'%(self.animal.name,self.__t))
    self.animal.getSpeedBehavior()
```

## 4 解决找不到模块的问题

我们知道Python中使用**import**导入需要的包，然而平时使用像vscode, pycharm这列ide时，经常出现找不到包的问题，错误信息如下：

```
Exception has occurred: ModuleNotFoundError
No module named 'animals'
```

要想解决此问题，需要首先了解Python的import机制。

当导入模块时，解释器会按照 `sys.path` 列表中的目录顺序来查找导入文件。要想查看解释器目前查找的目录顺序，先导入通过sys模块，使用sys.path，如下是import时查看的目录顺序：

```
['/home/zglg/mywork/md...mo/animals',
'/home/zglg/anaconda3...thon37.zip',
'/home/zglg/anaconda3.../python3.7',
'/home/zglg/anaconda3...ib-dynload',
'/home/zglg/anaconda3...e-packages']
```

看到animals包不在解释器要查找的目录里，所以出错了。

所以需添加animals包所在的文件夹路径，其中一种修改方法如下，直接粗暴向 `sys.path` 中添加找不到的目录：

```
# 调整为根目录(调用dirname一次获得其所在文件夹)
# 就当前文件目录，我们两次便定位到根目录 classdemo
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# __file__获取执行文件相对路径，整行为取上一级的上一级目录
sys.path.append(BASE_DIR)
import animals
```

再次启动程序，看到animals包目录已经显示搜索path列表中：

```
['/home/zglg/mywork/mdfiles/classdemo',
'/home/zglg/mywork/md...mo/animals',
'/home/zglg/anaconda3...thon37.zip',
'/home/zglg/anaconda3.../python3.7',
'/home/zglg/anaconda3...ib-dynload',
'/home/zglg/anaconda3...e-packages']
```

接下来就可以正常导入animals包，找不到包的问题解决。

以上就是此专题介绍的Python包、模块概念，以及如何应用到我们自己的实际项目的代码框架中，写出更加容易维护、可读性更好的代码。