

## 目录

1 变量名的命名规则 .....	7
2 Python 三种常用数据类型 .....	8
2.1 逻辑型 (Logical) .....	8
2.1.1 逻辑型数据运算规则.....	8
2.2 数值型 (Numeric) .....	8
2.2.1 数值型数据运算规则.....	8
2.2.2 数值型数据运算需要注意的地方.....	8
2.3 字符型 (Character) .....	9
3 python 中基本的数据类型 .....	10
4 序列：列表、元组、字符串、字节.....	10
4.1 序列概念与分类.....	10
4.2 序列通用操作：索引、切片、迭代、长度、运算.....	10
4.3 元组→不可变序列 .....	10
4.3.1 元组特点.....	10
4.3.2 定义元组时，也可以不使用小括号，只使用逗号分隔即可.....	11
4.3.3 元组的计算.....	11
4.3.4 元组支持两个方法 count() 与 index().....	12
4.3.5 元组存在的意义.....	12
5 字符串 (string) .....	13
5.1 字符串的 3 种创建方式.....	13
5.1.1 用单引号( ' ')、双引号( " ")创建字符串 .....	13
5.1.2 连续 3 个单引号或者 3 个单引号，可以帮助我们创建多行字符串 .....	13
5.1.3 空字符串和 len 函数 .....	13
5.2 常见的转义字符.....	14
5.2.1 '\n':换行符.....	14
5.2.2 '\':单引号.....	14
5.2.3 '\':在行尾时，续行符.....	14
5.2.4 '\t':表示空四个字符，也称缩进，相当于按一下 Tab 键.....	15
5.2.5 '\n\t':换行加每行空四格.....	15
5.3 字符串索引(切片).....	16
5.4 字符串分割.....	16
5.4.1 split()方法.....	16
5.5 字符串拼接.....	17
5.5.1 join()方法.....	17
5.5.2 “+”号拼接法.....	17
5.5.3 join 方法和 “+” 字符串拼接性能比较 .....	17
5.6 字符串驻留机制.....	18
5.6.1 字符串驻留机制定义 .....	18
5.7 字符串比较.....	19
5.8 字符串中常用函数.....	20
5.9 字符串中需要注意的地方.....	21
6 列表→可变序列 .....	26

6.1	列表的 5 种创建方式.....	26
6.1.1	用 [] 创建列表.....	26
6.1.2	用 list 创建列表.....	26
6.1.3	用 range 创建整数列表.....	26
6.1.4	用列表推导式创建列表.....	26
6.1.5	用 list 和 [] 创建空列表.....	27
6.2	列表元素的 5 中添加方式.....	27
6.2.1	append() 方法: 真正的在尾部添加元素, 速度最快。推荐使用。.....	27
6.2.2	extend() 方法: 将 B 列表元素添加到 A 列表。推荐使用, A.extend(B) .	27
6.2.3	insert() 方法.....	28
6.2.4	“+” 操作符扩展列表.....	28
6.2.5	“*” 操作符扩展列表.....	28
6.3	列表元素的 3 种删除方式.....	29
6.3.1	del 方法.....	29
6.3.2	pop() 方法: 括号中传入的是索引.....	29
6.3.3	remove() 方法: 删除列表中首次出现的元素.....	30
6.3.4	clear() 方法: 删除列表中所有元素.....	30
6.4	列表元素的索引.....	30
6.4.1	列表合法索引范围: [-列表长度, 列表长度-1].....	30
6.4.2	index() 方法: 传入的是列表元素.....	31
6.5	列表元素计数与长度.....	31
6.5.1	count(): 获取指定元素在列表中出现的次数.....	31
6.5.2	len(): 获取列表长度.....	31
6.6	列表切片: 操作列表一个区间的元素.....	32
6.6.1	切片语法: [start:end:step].....	32
6.6.2	通过切片修改对应区间元素.....	32
6.7	列表排序.....	33
6.7.1	sort() 方法: 原地修改列表的排序方法.....	33
6.7.2	sorted() 方法: 建立新列表的排序方法.....	33
6.7.3	reverse(): 列表反转, 不排序.....	34
6.7.4	列表中其他内置函数汇总.....	34
6.8	列表遍历.....	35
6.9	列表嵌套: 列表中元素, 还是一个列表.....	35
6.10	列表推导式.....	35
6.11	列表赋值、浅拷贝、深拷贝.....	36
6.11.1	赋值.....	36
6.11.2	浅拷贝.....	37
6.11.3	深拷贝.....	38
6.12	列表中需要注意的地方.....	39
7	字典.....	40
7.1	字典的 5 种创建方法.....	40
7.1.1	用 {} 创建字典.....	40
7.1.2	用 dict 创建字典.....	40
7.1.3	用 zip 函数创建字典.....	40

7.1.4 用 {}, dict 创建空字典.....	40
7.1.5 用 fromkeys 创建‘值为空’的字典.....	41
7.2 字典中元素访问方法.....	41
7.2.1 通过“键”获取“值”。若“键”不存在，则抛出异常.....	41
7.2.2 get() 方法：强烈推荐的字典元素访问.....	41
7.2.3 用 items 获取‘所有的键值对’.....	42
7.2.4 列出所有有‘键’：keys, 列出所有有‘值’：values.....	42
7.2.5 字典键值对的个数：len().....	43
7.3 向字典中“添加元素”.....	43
7.3.1 “键”存在，则覆盖原有“键值对”。“键”不存在，新增键值对.....	43
7.3.2 使用 update 把 b 字典的所有‘键值对’添加到 a 字典中.....	43
7.4 “删除”字典中元素.....	44
7.4.1 del 方法：删除指定的“键值对”.....	44
7.4.2 clear 方法：删除字典中所有的‘键值对’.....	44
7.4.3 pop 方法：删除指定的‘键’.....	44
7.4.4 popitem 方法：随机删除和返回‘键值对’.....	44
7.5 序列解包：运用于字典（类似于赋值）.....	45
7.5.1 利用 items：把‘键值对’赋给 b, c, d, e, f... ..	45
7.5.2 利用 keys：把‘键’赋给 g, h, i, j, k... ..	45
7.5.3 利用 values：把‘键’赋给 l, m, n, o, p... ..	45
8 格式化输出：%s 和 format 的用法.....	46
8.1 python 格式化历史起源.....	46
8.2 基本格式化(位置格式化).....	46
8.3 填充和对齐.....	46
8.3.1 填充.....	46
8.3.2 对齐.....	47
8.3.3 截断.....	48
8.3.4 填充和截断结合使用.....	48
8.4 和数字相关的语法.....	49
8.4.1 填充整数，使用 d.....	49
8.4.2 填充浮点数，使用 f.....	50
8.4.3 数字填充问题.....	51
8.4.4 修改填充符号.....	52
8.4.5 数字的正负号问题.....	52
8.4.6 d 和 f 前面有一个空格作用：保持对齐.....	53
9 集合(Set).....	54
9.1 集合的作用.....	54
9.2 集合的特征.....	54
9.3 集合中提供的方法.....	56
9.3.1 add()：向集合中添加指定的元素.....	56
9.3.2 remove()：删除集合中指定的元素。如果元素不存在，报错.....	56
9.3.3 discard()：删除集合中指定的元素。如果元素不存在，不进行任何操作.....	56
9.3.4 pop()：删除并返回任意集合中元素——一般不用.....	57
9.3.5 clear()：删除集合中的所有元素.....	57

9.3.6	<code>copy()</code> : 对集合进行复制。(深拷贝? 浅拷贝)	57
9.3.7	<code>difference()</code> : 两个集合的差集, 产生新的集合, 但不改变当前集合..	58
9.3.8	<code>difference_update()</code> : 功能与 <code>difference()</code> 相同, 但改变当前集合 ..	58
9.3.9	<code>intersection()</code> : 两个集合的交集, 产生新的集合, 但不改变当前集合	58
9.3.10	<code>intersection_update()</code> : 功能与 <code>difference()</code> 相同, 但改变当前集合	59
9.3.11	<code>union()</code> : 两个集合的并集, 产生新的集合, 但不改变当前集合.....	59
9.3.12	<code>update()</code> : 功能与 <code>union()</code> 相同, 但改变当前集合 .....	59
9.3.13	<code>symmetric_difference()</code> : 两个集合的对称差集.....	60
9.3.14	<code>symmetric_difference_update()</code> : 功能与 <code>symmetric_difference()</code> 相同, 但改变当前集合.....	60
9.3.15	<code>isdisjoint()</code> : 判断当前集合与参数集合, 是否交集为空; 是返回 <code>True</code> , 否返回 <code>False</code> .....	60
9.3.16	<code>issubset()</code> : 判断当前集合是否为参数集合, 的子集; 是返回 <code>True</code> , 否返回 <code>False</code> .....	61
9.3.17	<code>issuperset()</code> : 判断当前集合是否为参数集合, 的父集; 是返回 <code>True</code> , 否返回 <code>False</code> .....	61
9.4	集合的运算.....	62
9.4.1	<code>in</code> : 判断某个元素是否在集合中, 是返回 <code>True</code> , 否返回 <code>False</code> .....	62
9.4.2	<code>&amp;</code> : 返回两个元素的交集, 相当于 <code>s.intersection(t)</code> .....	62
9.4.4	<code> </code> : 返回两个元素的并集, 相当于 <code>s.union(t)</code> .....	62
9.4.5	<code>-</code> : 返回两个元素的差集, 相当于 <code>s.difference(t)</code> .....	62
9.4.6	<code>^</code> : 返回两个元素的对称差集, 相当于 <code>s.symmetric_difference(t)</code> ...	63
9.5	集合比较运算 .....	63
9.5.1	<code>==</code> : 两个集合中元素是否一致.....	63
9.5.2	<code>&gt;</code> 、 <code>&gt;=</code> 、 <code>&lt;</code> 、 <code>&lt;=</code> : 父集、子集比较.....	63
9.5.3	<code>is</code> 、 <code>is not</code> : 比较两个集合是否是同一个对象.....	64
9.6	集合推导式: 可以自动去重.....	64
9.7	真值测试.....	65
10	异常.....	66
10.1	异常的概念.....	66
10.2	常见异常类型.....	67

# 1 Python 变量名的命名规则

## 1.1 什么是变量？

变量，用于在内存中存放程序数据的容器。计算机的核心功能就是“计算”，CPU 是负责计算的，而计算需要数据吧？数据就存放在内存里，例如：将黄同学的姓名，年龄存下来，让后面的程序调用。那么，怎么存呢？直接使用变量名=值，即可。

## 1.2 变量的使用规则

由于 Python 程序是从上到下依次执行的，所以在某个变量之前，必须先定义后调用，否则后面会报错。

## 1.3 变量命名规范

### ① 驼峰命名法

```
>>> FirstName = "黄"
>>> LastName = "伟"
```

### ② 下划线命名法

```
>>> first_name = "黄"
>>> last_name = "伟"
```

注意：第②种是推荐使用的，看起来更清晰。

## 1.4 变量名的定义

- ① 变量名可由 a-z, A-Z, 数字, 下划线(\_) 组成, 首字母不能为数字和下划线(\_);
- ② Python 对大小写敏感, 变量 a 和变量 A 表示不同的变量;
- ③ 变量名不能为 Python 中的保留字(自行百度“保留字”);

## 2 Python 三种常用数据类型

### 2.1 什么是数据类型？

人类有思想，很容易区分汉字和数字的区别，例如，你知道 1 是数字，“中国”是汉字。计算机虽然很强大，但是它没有思想，它不知道哪个是汉字，哪个是数字，除非你明确告诉它。这就是我们要说的“数据类型”，数据类型将它们进行了明确的划分，告诉计算机哪个是数字，那个是字符串。Python 中常用到的数据类型有逻辑型（Logical）、数值型（Numeric）、字符型（Character）。

### 2.2 逻辑型（Logical）

又叫做“布尔型”，用于只有两种取值(0 和 1，真和假)的场合，首字母是大写。

True      真

False     假

#### 2.2.1 逻辑型数据运算规则

&        与，两个逻辑型数据中，一假则为假。

|        或，两个逻辑型数据中，一真则为真。

not    非，not True 就是 False，not False 就是 True。

### 2.3 数值型（Numeric）

就是我们数学里面学过的实数：包括负数、0、正数。

#### 2.3.1 数值型数据运算规则

就是我们经常用到的加、减、乘、除。

#### 2.3.2 数值型数据运算需要注意的地方

下面几种情况，是我们需要额外注意的地方，我这里单独列举出来。

1) “/” 代表除法，“//” 代表取整；

```
>>> 7/4
```

```
1.75
```

```
>>> 7//4
1
```

## 2) “%” 表示求余

```
>>> 10 % 4
2
>>> 5 % 3
2
```

## 3) 一个关于浮点数需要注意的地方

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a+b) == 6.3
False
>>> from decimal import Decimal
>>> a = Decimal("4.2")
>>> b = Decimal("2.1")
>>> a+b
Decimal('6.3')
>>> (a+b) == Decimal("6.3")
True
>>> (a+b)/3
Decimal('2.1')
```

## 2.4 字符型 (Character)

字符型数据代表了所有可定义的字符，无运算规则。

### 3 python 中基本的数据结构

Python 常用数据结构如下，我们尤其需要注意字符串、列表和字典这 3 种。

String 字符串

List 列表

Tuple 元组

Set 集合

Dic 字典

### 4 序列：列表、元组、字符串、字节

#### 4.1 序列概念与分类

序列是一种可迭代对象，可以存储多个数据，并提供数据的访问。序列中的数据，称之为“序列元素”。Python 中内置的序列类型有：

列表 (list)

元组 (tuple)

字符串 (str)

字节 (bytes)

#### 4.2 序列的通用操作：索引、切片、迭代、长度、运算

- 1、索引：通过索引访问序列中指定位置的元素（单个）；
- 2、切片：通过切片访问序列中一个区间的元素（多个）；
- 3、迭代：序列作为可迭代对象，因此，可以通过 for 循环进行遍历；
- 4、长度：可以通过 len() 函数获取序列长度（序列中还有元素的个数）；
- 5、运算：序列支持+、\*、in、not in、比较、布尔运算符；

#### 4.3 元组→不可变序列

##### 4.3.1 元组特点

- 1、元组与列表相似。不同之处在于：元组不可修改；
- 2、元组也支持索引、切片、遍历、长度、运算；
- 3、Python 中使用 tuple 表示元组类型，并用 ( ) 定义元组；



```

>>> t = (1, 2, 3)
>>> type(t)
<class 'tuple'>
>>> a = (1)
>>> type(a)
<class 'int'>
>>> b = (1,)
>>> type(b)
<class 'tuple'>
>>> #当只有一个元素时，元素后面的逗号不能省略。
>>> #否则()会被解析为调解运算优先级的()，而不会解析为元组类型。

```

#### 4.3.2 定义元组时，也可以不使用小括号，只使用逗号分隔即可

以下情况，元组小括号不能省略：

```

>>> # 1 定义空元组
>>> x = ()
>>> # 2 元组嵌套
>>> x = ((1, 2), (3, 4))
>>> # 3 元组作为表达式一部分
>>> x = 1, 2 + 3, 4 不加括号 定义元组，特别不好理解
>>> x
(1, 5, 4)
>>> x = (1, 2) + (3, 4)
>>> x
(1, 2, 3, 4)
>>> # 建议：为了增强程序可读性，只要是使用元组，总是加上小括号。

```

#### 4.3.3 元组的计算

```

>>> (1, 2) + (3, 4) #加法运算
(1, 2, 3, 4)
>>> (1, 2) * 2      #乘法运算
(1, 2, 1, 2)
>>> # in运算符
>>> 1 in (1, 2)
True
>>> 3 in (1, 2)
False
>>> # is运算符
>>> k = (1, 2)
>>> s = (1, 2)
>>> k is k
True
>>> k is s
False
>>> # =运算符
>>> k == s
True
>>> # >运算符
>>> a = (1, 2)
>>> b = (3, 4)
>>> b > a
True
>>> a > b
False
>>> # > < 就是将元组对应元素进行比较
>>> |

```

#### 4.3.4 元组索引、切片、遍历

```
>>> k = (1, 2, 3, 4, 5)
>>> #元组索引
>>> k[0]
1
>>> #元组切片，仍然是元组
>>> k[2:4]
(3, 4)
>>> #元组遍历
>>> for i in k:
>>>     print(i)

1
2
3
4
5
>>> #元组不可修改其中元素，否则，会报错
>>> k = (1, 2)
>>> k[1] = 12
Traceback (most recent call last):
  File "<pyshell#170>", line 1, in <module>
    k[1] = 12
TypeError: 'tuple' object does not support item assignment
```

#### 4.3.5 元组支持两个方法 count() 与 index()

```
>>> x = (1, 2, 3, 3, 1, 4, 2, 5)
>>> x.count(2)
2
>>> x.index(2)
1
>>> x.index(2, 3, 7)
6
>>> #具体函数意义，参考列表中方法详解
```

#### 4.3.6 元组存在的意义

元组作为一个不可变序列，可以保证数据在程序运行过程中，不会意外由自己或他人修改（例如在函数参数传递中），这样可以保证数据的完整性。

## 5 字符串 (string)

### 5.1 字符串的 3 种创建方式

#### 5.1.1 用单引号( ' ')、双引号( " ")创建字符串

```
>>> a = 'I am a student'
>>> print(a)
I am a student
>>> b = "I am a teacher"
>>> print(b)
I am a teacher
```

#### 5.1.2 连续 3 个单引号或者 3 个双引号，可以帮助我们创建多行字符串

```
>>> a = '''
I am a student
My name is 黄伟
I am a teacher
My name is 陈丽
'''
>>> print(a)
I am a student
My name is 黄伟
I am a teacher
My name is 陈丽
```

#### 5.1.3 空字符串和 len 函数

# len: 用于计算字符串里面含有多少个字符。

```
>>> c = ''
>>> print(len(c))
0
```

## 5.2 常见的转义字符

Python 中常见的转义字符，我这里为大家详细列举出来了。

'\n': 换行符;

'\'' : 单引号;

'\': 在行尾时，续行符;

'\t': 空四个字符，也称缩进，相当于按一下 Tab 键;

'\n\t': 换行加每行空四格;

### 5.2.1 '\n': 换行符

```
>>> a = 'i\nlove\nyou'
>>> print(a)
i
love
you
```

### 5.2.2 '\'' : 单引号

```
>>> s = 'Yes,he doesn't'
SyntaxError: invalid syntax
>>> s = 'Yes,he doesn\'t'
>>> print(s, type(s), len(s))
Yes,he doesn't <class 'str'> 14
```

### 5.2.3 '\': 在行尾时，续行符

```
>>> s = 'abcd\
eegf'
>>> print(s)
Abcdeegf
```

5.2.4 '\t': 表示空四个字符, 也称缩进, 相当于按一下 Tab 键

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

5.2.5 '\n\t': 换行加每行空四格

```
>>> print("\n\tHuangwei\n\tlove\n\tChenli")

    Huangwei
    love
    Chenli
>>> print("\tHuangwei\n\tlove\n\tChenli")

    Huangwei
    love
    Chenli
```

注: \t\n 效果和\n 是一样的, 不建议使用。

```
>>> print("\t\nPython")

Python
>>> print("\nPython")

Python
>>> print("\n\tPython")

    Python
```

### 5.3 字符串索引(切片)

```
>>> x = 'python'
>>> print(x[0],x[5])
p n
>>> print(x[-6],x[-1])
p n
>>> print(x[1:4])
yth
>>> print(x[-5:-2])
yth
>>> x[::-1]
'nohtyp'
>>> x[::-2]
'nhy'
```

### 5.4 字符串分割

#### 5.4.1 split() 方法

```
>>> a = 'to be or not to be'
>>> print(a.split())
['to', 'be', 'or', 'not', 'to', 'be']
>>> c = 'to be or not to be'
>>> print(c.split('be'))
['to ', ' or not to ', '']
```

**注：**`split` 可以基于指定分隔符，将字符串分割成为多个字符串，然后存到一个“列表”中(!!!! 很关键，这个)。如果不指定分隔符，将默认使用空白字符（空格/换行符/制表符）。

## 5.5 字符串拼接

### 5.5.1 join() 方法

```
>>> a = ['陈', '丽', '美', '丽']
>>> b = ''.join(a)
>>> print(b)
陈丽美丽
>>> c = '\n'.join(a)
>>> print(c)
陈
丽
美
丽
```

### 5.5.2 “+” 号拼接法

```
>>> print('str' + 'ing')
string
>>> str = ''
>>> for i in range(5):
    str += '陈丽'    # 相当于 str = str + '陈丽'
>>> print(str)
陈丽陈丽陈丽陈丽陈丽
```

### 5.5.3 join 方法和 “+” 字符串拼接性能比较

```
import time

time01 = time.time()
st = ''
for i in range(1000000):
    st += '陈丽'
time02 = time.time()
```

```

print('用时: ' + str(time02-time01))
用时: 1.2170696258544922

time03 = time.time()
a = []

#随着迭代次数增大。我们发现：使用 join 内置函数，拼接字符串所用时间少。
for i in range(1000000):
    a.append('陈丽')
li = ''.join(a)
time04 = time.time()
print('用时: ' + str(time04-time03))
用时: 0.24601387977600098

```

综上所述：进行字符串拼接，推荐使用‘join’进行字符串拼接，少用‘+’。

## 5.6 字符串驻留机制

### 5.6.1 字符串驻留机制定义

Python 支持字符串驻留机制，对于符合标识符的字符串（注意：仅仅包含下划线\_、字母、数字），才会启用字符串驻留机制。此时，保存一份相同且不可变的字符串，不同的值被存在驻留池中，因此，他们还是同一个东西。

```

>>> m = '12_abv'
>>> n = '12_abv'
>>> print(id(m), id(n))
51210312    51210312
>>> print(m == n)
True
# m 和 n 是同一个对象，存储地址相同才是同一个对象。
>>> print(m is n)
True

-----

>>> p = 'ab#'

```



```

>>> q = 'ab#'
>>> print(id(p), id(q))
51209808 51210368
>>> print(p == q)
True
# p 和 q 不是同一个对象，存储地址相同才是同一个对象。
>>> print(p is q)
False

```

## 5.7 字符串比较

`==`、`!=`: 比较字符串 a 和字符串 b，**是否含有相同的字符**。

`is`、`not is`: 比较两个对象，**是否是同一个对象**。

`in`、`not in`: 判断某个字符，**是否存在于某个字符串中**。

```

>>> str1 = 'abcdefgh'
>>> print('a' in str1)
True
>>> print('ab' in str1)
True
>>> str2 = "123"
>>> str3 = "123"
>>> str2 == str3
True
>>> str2 is str3
True
>>> id(str2)
1995935692312
>>> id(str3)
1995935692312

```

## 5.8 字符串中常用函数

**注意：“字符串”中，空白符也算是真实存在的一个字符。**

### 5.8.1 find() 函数

功能：检测字符串是否包含指定字符。如果包含指定字符，则返回开始的索引；否则，返回-1。

```
>>> st = "hello world"
>>> st.find("or")
7
>>> st.find("ww")
-1
```

### 5.8.2 index() 函数

功能：检测字符串是否包含指定字符。如果包含指定字符，则返回开始的索引；否则，提示 ValueError 错误。

```
>>> st = "hello world"
>>> st.index("or")
7
>>> st.index("ww")
ValueError                                Traceback (most recent call 1
ast)
<ipython-input-9-4958b3271b1c> in <module> ()
----> 1 st.index("ww")

ValueError: s
```

### 5.8.3 count() 函数

功能：统计字符串中，某指定字符在指定索引范围内，出现的次数。索引范围也是：左闭右开区间。

注意：如果不指定索引范围，表示在整个字符串中，搜索指定字符出现的次数。

```
>>> st = "hello world"
>>> st.count("l")

>>> st.count("l", 2, len(st))
3
>>> st.count("l", 3, len(st))
2
>>> st.count("l", 2, 3)
1
```

### 5.8.4 replace 函数

语法：st.replace(str1, str2, count)。

功能：将字符串 st 中的 str1 替换为 str2。

注意：如果不指定 count，则表示整个替换；如果指定 count=1，则表示只替换一次，count=2，则表示只替换两次。

```
>>> st = "hello world"
>>> st.replace("l", "6")
'he66o wor6d'
>>> st.replace("l", "6", 1)
'he6lo world'
>>> st.replace("l", "6", 2)
'he66o world'
>>> st.replace("l", "6", 3)
```

```
'he66o wor6d'  
>>> st.replace("l","6",100)
```

### 5.8.5 split() 函数

语法: `st.split('分隔符', maxSplit)`

功能: 将字符串按照指定分隔符, 进行分割。

注意: 如果 `split` 中什么都不写, 则默认按照空格进行分割; 如果指定了分隔符, 则按照指定分隔符, 进行分割。

`maxSplit` 作用: 不好叙述, 自己看下面的例子就明白。

```
>>> st = "hello world"  
>>> st.split()  
['hello', 'world']  
>>> st.split("l")  
['he', '', 'o wor', 'd']  
>>> st.split("l",1)  
['he', 'lo world']  
>>> st.split("l",2)  
['he', '', 'o world']  
>>> st.split("l",3)  
['he', '', 'o wor', 'd']  
>>> st.split("l",100)  
['he', '', 'o wor', 'd']
```

### 5.8.6 startswith()

语法: `st.startswith(str1)`

功能: 检查字符串 `st` 是否以字符串 `str1` 开头, 若是, 则返回 `True`; 否则, 返回 `False`。

```
>>> st = "hello world"  
>>> st.startswith("h")  
True  
>>> st.startswith("he")  
True
```

```
>>> st.startswith("hel")
True
```

#### 5.8.7 `endwith()`

语法: `st.endswith(str1)`

功能: 检查字符串 `st` 是否以字符串 `str1` 结尾, 若是, 则返回 `True`; 否则, 返回 `False`。

```
>>> st = "hello world"
>>> st.endswith("d")
True
>>> st.endswith("ld")
True
>>> st.endswith("rld")
True
>>> st.endswith("rd")
False
```

#### 5.8.8 `lower()`

语法: `st.lower()`

功能: 将字符串的所有字母转换为小写。

```
>>> st = "Huang Wei"
>>> st.lower()
'huang wei'
```

#### 5.8.9 `upper()`

语法: `st.upper()`

功能: 将字符串的所有字母转换为大写。

```
>>> st = "Huang Wei"
>>> st.upper()
'HUANG WEI'
```

### 5.8.10 strip()

语法: `st.strip()`

功能: 去掉字符串左右两边的空白字符。

```
>>> st = " Huang Wei"
>>> st.strip()
'Huang Wei'
>>> st1 = "Huang Wei  "
>>> st1.strip()
'Huang Wei'
>>> st2 = st = " Huang Wei "
>>> st2.strip()
'Huang Wei'
```

**注意:** `st.rstrip()` : 去掉字符串右边的空白字符。

**注意:** `st.lstrip()` : 去掉字符串左边的空白字符。

### 5.8.11 join()

语法: `st.join(str1)`

功能: 在指定字符串 `str1` 中, 每相邻元素中间插入 `st` 字符串, 形成新的字符串。

注意: 是在 `str1` 中间插入 `st`, 而不是在 `st` 中间插入 `str1`。

```
>>> st = "Huang Wei"
>>> str1 = "abc"
>>> st.join(str1)
'aHuang WeibHuang Weic'
```

### 5.8.12 isalpha()

语法: str.isalpha()

功能: 如果字符串 str 中只包含字母, 则返回 True; 否则, 返回 False。

注意: 只有字符串中全部是字母, 才会返回 True, 中间有空格都不行。

```
>>> st = "haung wei"
>>> st.isalpha() # 因为还有空格, 所以返回 false。
False
>>> st1 = "haungwei"
>>> st1.isalpha()
True
```

### 5.8.13 isdigit()

语法: str.isdigit()

功能: 如果字符串 str 中只包含数字, 则返回 True; 否则, 返回 False。

```
>>> st = "123897"
>>> st.isdigit()
True
>>> st1 = "123 897"
>>> st1.isdigit()
False
```

## 5.9 字符串中需要注意的地方

字符串本质: 字符序列。

Python 字符串是不能变的的。因此, 我们无法对字符串进行修改。

Python 不支持单字符。就算是单字符, 也作为字符串来使用。

## 6 列表→可变序列

### 6.1 列表的 5 种创建方式

#### 6.1.1 用 [] 创建列表

```
>>> a = [1, 2, 3]
>>> print(a, type(a))
[1, 2, 3] <class 'list'>
```

#### 6.1.2 用 list 创建列表

```
>>> b = list('abc')
>>> b, type(b)
['a', 'b', 'c'] <class 'list'>
>>> c = list((1, 2, 3))
>>> c
[1, 2, 3]
#对于字典，生成的是键 key 列表。
>>> d = list({"aa":1, "bb":3})
>>> d
['aa', 'bb']
```

#### 6.1.3 用 range 创建整数列表

```
>>> e = list(range(10))
>>> print(e, type(e))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>
>>> ee = list(range(3, -10, -3))
>>> print(ee, type(ee))
[3, 0, -3, -6, -9] <class 'list'>
```

#### 6.1.4 用列表推导式创建列表

```
>>> f = [i for i in range(5)]
>>> print(f, type(f))
[0, 1, 2, 3, 4] <class 'list'>
```



### 6.1.5 用 list 和 [] 创建空列表

```
>>> g = list()
>>> print(g)
[]
>>> h = []
>>> print(h)
[]
```

## 6.2 列表元素的 5 中添加方式

6.2.1 append() 方法：真正的在尾部添加元素，速度最快。推荐使用。

注：我们发现在尾部添加元素后，b 的存储地址没有改变。原地操作列表。

```
>>> a = ['黄伟', '陈丽', 'QQ', '微信']
>>> b = []
>>> print(id(b))
51115016
>>> for i in a:
        b.append(i)
>>> print(b)
['黄伟', '陈丽', 'QQ', '微信']
>>> print(id(b))
51115016
```

6.2.2 extend() 方法：将 B 列表元素添加到 A 列表。推荐使用，A.extend(B)

注：将一个列表的元素，添加到另外一个列表元素的尾部。添加元素前后，地址不变。

```
>>> c = [1, 2, 3]
>>> print(id(c))
51296456
>>> d = ['黄伟', '陈丽']
>>> c.extend(d)
```

```
>>> print(c, id(c))  
[1, 2, 3, '黄伟', '陈丽'] 51296456
```

### 6.2.3 insert() 方法

**注：**在列表指定位置插入元素。

当涉及大量元素时，尽量避免使用。因为，会让插入位置后面的元素进行大面积移动，影响处理速度。

**类似这样的函数还有：**remove、pop、del。

### 6.2.4 “+” 操作符扩展列表

```
>>> a = [20, 40]  
>>> print(id(a))  
51297096  
>>> a = a + [50]  
>>> print(a)  
[20, 40, 50]  
>>> print(id(a))  
51296968
```

**注：**因为存储地址前后发生改变，属于创建了新列表，增加了内存。**不建议使用。**

### 6.2.5 “\*” 操作符扩展列表

```
>>> a = [1, 2, 3]  
>>> print(id(a))  
51297416  
>>> b = a * 3  
>>> print(b)  
[1, 2, 3, 1, 2, 3, 1, 2, 3]  
>>> print(id(b))  
51225352
```

**注：****不建议使用。**（字符串、元组也可以用‘乘法扩展’，增加相同的元素）

## 6.3 列表元素的 3 种删除方式

### 6.3.1 del 方法

注：删除列表指定位置的元素。

```
>>> a = [1, 2, 3]
>>> del a[0]
>>> print(a)
[2, 3]
#删除整个列表
>>> del a
>>> a
Traceback (most recent call last):
  File "<pyshell#417>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

### 6.3.2 pop() 方法：括号中传入的是索引

注：删除指定位置元素，并返回指定位置元素。若不指定位置，默认删除列表末尾元素。推荐使用：不指定索引的用法。

```
>>> b = [1, 2, 3]
>>> b.pop(1)
2
>>> print(b)
[1, 3]

>>> c = ['a', 'b', 'c']
>>> c.pop()
'c'
>>> c.pop()
'b'
>>> c.pop()
```

```
'a'
>>> print(c)
[]
>>> c.pop()
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    c.pop()
IndexError: pop from empty list
```

### 6.3.3 remove() 方法：删除列表中首次出现的元素

注：若列表中，有重复元素，则删除列表中，首次出现的该元素。若删除元素不存在，则抛出异常。

```
>>> d = [1, 2, 5, 2, 3, 4, 5, 1, 3, 2, 1]
>>> d.remove(2)
>>> d
[1, 5, 2, 3, 4, 5, 1, 3, 2, 1]
```

### 6.3.4 clear() 方法：删除列表中所有元素

```
>>> d = [1, 2, 5, 2, 3, 4, 5, 1, 3, 2, 1]
>>> d.clear()
>>> d
[]
```

## 6.4 列表元素的索引

### 6.4.1 列表合法索引范围：[-列表长度, 列表长度-1]

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[4]
5
>>> a[9]
Traceback (most recent call last):
```

```
File "<pyshell#28>", line 1, in <module>
    a[9]
IndexError: list index out of range
```

#### 6.4.2 index() 方法：传入的是列表元素

**语法：**index(指定元素, start, end), start、end 表示从哪个位置开始进行搜索。

**注：**传入列表元素，返回该元素的索引

```
>>> b = [12, 13, 14, 12, 12, 14, 13, 12, 14, 15]
>>> b.index(13)
1
>>> b.index(13, 2)
6
>>> b.index(20)
Traceback (most recent call last):
  File "<pyshell#397>", line 1, in <module>
    b.index(20)
ValueError: 20 is not in list
```

### 6.5 列表元素计数与长度

#### 6.5.1 count()：获取指定元素在列表中出现的次数

```
>>> a = [12, 13, 14, 12, 12, 14, 13, 12, 14, 15]
>>> a.count(14)
3
```

#### 6.5.2 len()：获取列表长度

```
>>> b = [12, 13, 14, 12, 12, 14, 13, 12, 14, 15]
>>> len(b)
10
```

## 6.6 列表切片：操作列表一个区间的元素

### 6.6.1 切片语法：[start:end:step]

start: 指定切片起始点;

end:指定切片终止点;

step: 指定步长值, 默认为 1

注意 1: 切片区间包含起始点, 不包含终止点;

注意 2: start、end、step 都是可选的;

```
>>> li = [5, 3, 8, 6, 10, 9, -2]
>>> li[0:3]
[5, 3, 8]
>>> #步长指定增量值
>>> li[0:6:2]
[5, 8, 10]
>>> #start、end、step也可以是负值
>>> #如果切片的方向与增量的方向相反, 则无法获取任何元素, 得到空列表[]
>>> li[-1:-4]
[]
>>> li[-1:-4:-1]
[-2, 9, 10]
>>> li[-4:-1]
[6, 10, 9]

>>> #省略start或end
>>> #增量为正, 切片从左往右; 增量为负, 切片从右往左
>>> li = [5, 3, 8, 6, 10, 9, -2]
>>> li[:4]
[5, 3, 8, 6]
>>> li[:4:1]
[5, 3, 8, 6]
>>> li[:4:-1]
[-2, 9]
>>> li[4:]
[10, 9, -2]
>>> li[4::-1]
[10, 6, 8, 3, 5]
```

### 6.6.2 通过切片修改对应区间元素

```
>>> li = [5, 3, 8, 6, 10, 9, -2]
>>> #列表切片返回时列表类型, 因此, 赋值时, 需要指定一个列表
>>> li[2:5] = [100, 101, 102]
>>> li
[5, 3, 100, 101, 102, 9, -2]
>>> li[2:5] = [] #相当于删除列表中指定元素
>>> li
[5, 3, 9, -2]
>>> li[2:2] = [32, 50] #相当于插入元素
>>> li
[5, 3, 32, 50, 9, -2]
```

## 6.7 列表排序

### 6.7.1 sort() 方法：原地修改列表的排序方法。

```
>>> a = [20, 10, 40, 30]
>>> print(id(a))
51969480
>>> a.sort()
>>> print(a)
[10, 20, 30, 40]
>>> print(id(a))
51969480

>>> b = [5, 2, 3, 1, 5, 4]
>>> b.sort(reverse=True)    #降序
>>> b
[5, 5, 4, 3, 2, 1]
```

**注 1：**“默认是升序”，参数 `reverse=True`，表示将列表降序。

**注 2：**“原地修改列表”，不建立新列表的排序方法。意思就是说：排序前后，列表存储地址不会变，使用 `id` 函数，查看前后列表地址。

**注 3：**`a.sort` 不能用一个新的变量去接收，因为列表前后地址没变，是在原地修改列表。所以不可能用新的变量接收。`append` 也是类似用法，不能用新的变量去接收。

### 6.7.2 sorted() 方法：建立新列表的排序方法

```
>>> c = [20, 10, 40, 30]
>>> print(id(c))
52034312
>>> d = sorted(c)
>>> print(d)
[10, 20, 30, 40]
>>> print(id(d))
```

```
52033992
```

```
>>> e = [5, 2, 3, 1, 5, 4]
>>> f = sorted(e, reverse=True)
>>> f
[5, 5, 4, 3, 2, 1]
```

**注 1：**sorted 方法，属于内置函数。而不属于列表的方法。因此是：sorted(a) 而不是 a.sorted()。

**注 2：**sorted 方法，重新建立新列表的排序，排序不是对本列表的排序，本列表并不会变，而是将排序后的列表存放在其他地址中。因此“可以用新变量接收”。

**注 3：**“默认是升序”，参数 reverse=True，表示将列表降序。

### 6.7.3 reverse()：列表反转，不排序

```
>>> x = [1, 4, 2, 3, 5]
>>> x.reverse()
>>> x
[5, 3, 2, 4, 1]
```

### 6.7.4 列表中其他内置函数汇总

max()：返回列表中的最大值

min()：返回列表中的最小值

sum()：对数值型列表中，所有元素进行求和，有非数值型数据会报错。

```
>>> e = [3, 4, 12, 7, 9, 6]
>>> max(e)
12
>>> min(e)
3
>>> sum(e)
41
```



## 6.8 列表遍历

```
>>> x = ["赵", "钱", "孙", "李", "周"]
>>> index = 0
>>> while index <= len(x)-1:
    print(x[index])
    index += 1

赵
钱
孙
李
周

>>> x = ["赵", "钱", "孙", "李", "周"]
>>> for i in x:
    print(i)

赵
钱
孙
李
周
```

注：遍历列表。For 循环比 while 循环更好。

## 6.9 列表嵌套：列表中元素，还是一个列表

```
>>> x = [[1, 2, 3]]
>>> for i in x:
...     for j in i:
...         print(j)
...
1
2
3
```

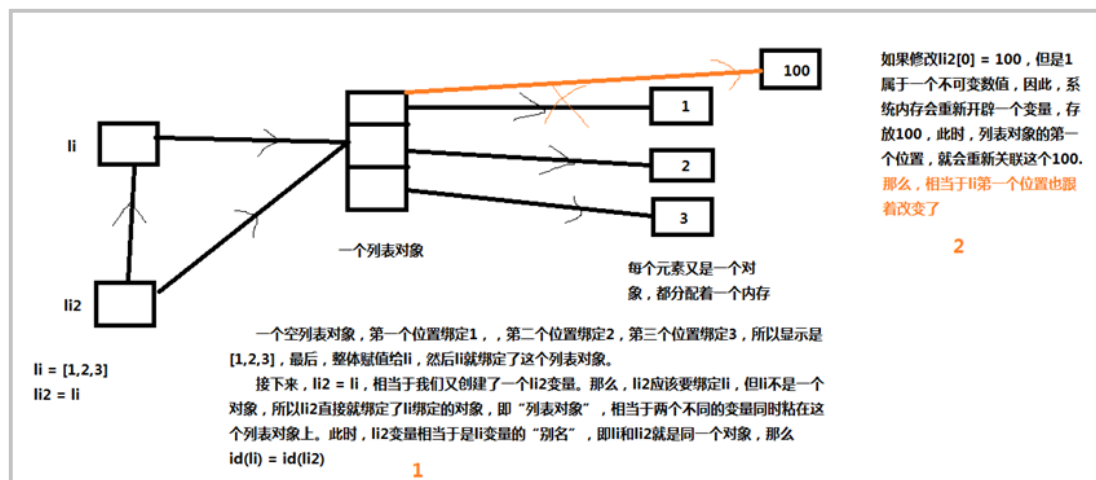
## 6.10 列表推导式

```
>>> r = [[i, j] for i in range(1, 6) if i != 1 and i != 4 for j in range(1, 6) if j != 2 and j != 5]
>>> r
[[2, 1], [2, 3], [2, 4], [3, 1], [3, 3], [3, 4], [5, 1], [5, 3], [5, 4]]
>>> r = [(i, j) for i in range(1, 6) if i != 1 and i != 4 for j in range(1, 6) if j != 2 and j != 5]
>>> r
[(2, 1), (2, 3), (2, 4), (3, 1), (3, 3), (3, 4), (5, 1), (5, 3), (5, 4)]
>>> i
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    i
NameError: name 'i' is not defined
>>> j
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    j
NameError: name 'j' is not defined
```

列表推导式中的变量，只在列表推导式中使用，在列表推导式之外无效，因此，在列表推导式中的变量，不会影响到推导式之外。但是，在 for 循环中定义的变量，在 for 循环之外仍然有效。

## 6.11 列表赋值、浅拷贝、深拷贝

### 6.11 赋值

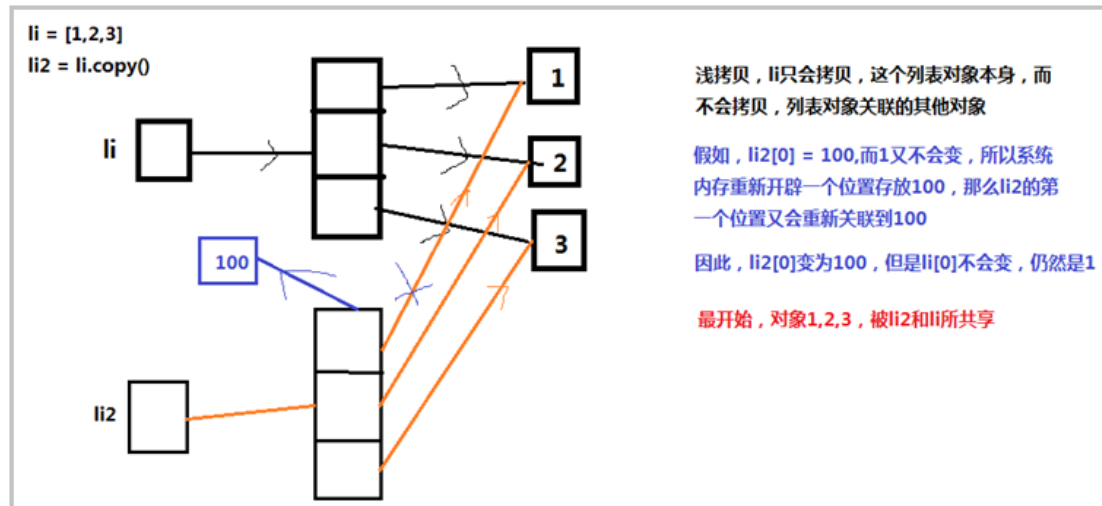


演示如下：

```
>>> li = [1, 2, 3]
>>> li2 = li
>>> id(li) == id(li2)
True
>>> li2[0] = 100
>>> li
[100, 2, 3]
```

## 6.12 浅拷贝

list（列表）的 copy 方法，实现的是浅拷贝。仅仅拷贝当前对象，而不会拷贝当前对象内部所关联的对象。

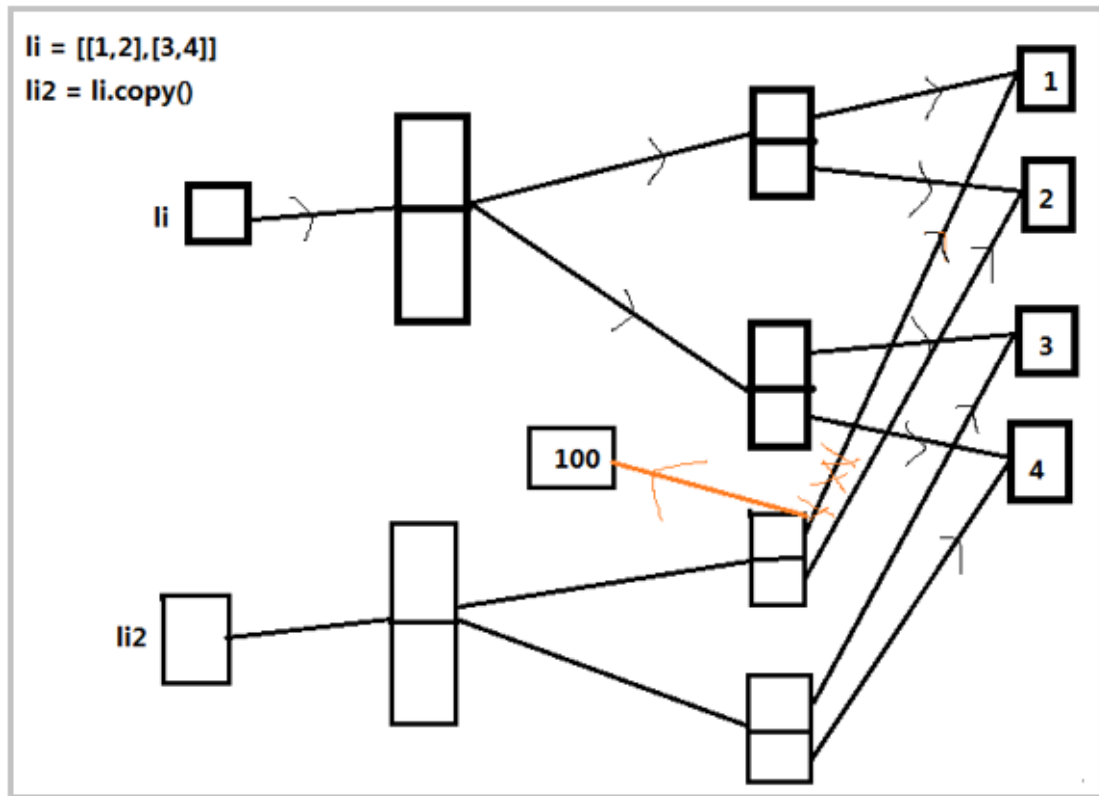


演示如下：

```
>>> li = [1, 2, 3]
>>> li2 = li.copy()
>>> id(li) == id(li2)
False
>>> li2[0] = 100
>>> li2
[100, 2, 3]
>>> li
[1, 2, 3]
```

### 6.13 深拷贝

如果列表中元素是可变类型，此时，浅拷贝无法实现期望的效果。若期望 li2 的改变不影响 li，我们可以采用“深拷贝”操作。“深拷贝”不仅拷贝当前对象本身，同时还会拷贝当前对象，所关联的可变对象，一直拷贝到没有可变对象为止。



演示如下：

```
>>> import copy
>>> li = [[1,2],[3,4]]
>>> li2 = copy.copy(li)
>>> li2[0][0] = 666
>>> li
[[666, 2], [3, 4]]
>>> li2
[[666, 2], [3, 4]]

-----

>>> li = [[1,2],[3,4]]
>>> li2 = copy.deepcopy(li)
```

```
>>> li2[0][0] = 666
>>> li
[[1, 2], [3, 4]]
>>> li2
[[666, 2], [3, 4]]
```

## 6.12 列表中需要注意的地方

**注 1：**列表元素可以各不相同，可以是任意类型。

```
>>> lis = [1, 2.0, False, "abc"]
>>> lis
[1, 2.0, False, 'abc']
```

**注 2：**列表大小可变，可以根据需要，随时增加或者缩小列表。

**注 3：**‘字符串’与‘列表’都是‘序列类型’，只不过“字符串”是字符序列，“列表”是任何元素的序列。

## 7 字典

### 7.1 字典的 5 种创建方法

#### 7.1.1 用 {} 创建字典

```
>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> print(a)
{'name': '陈丽', 'age': 18, 'job': 'teacher'}

>>> b = {'name': '陈丽', 'age': 18, 'job': ['teacher', 'wife']}
>>> print(b)
{'name': '陈丽', 'age': 18, 'job': ['teacher', 'wife']}
```

#### 7.1.2 用 dict 创建字典

```
>>> c = dict(name='张伟', age=19)
>>> print(c)
{'name': '张伟', 'age': 19}

>>> d = dict([('name', '李丽'), ('age', 18)])
>>> print(d)
{'name': '李丽', 'age': 18}
```

#### 7.1.3 用 zip 函数创建字典

```
>>> x = ['name', 'age', 'job']
>>> y = ['陈丽', '18', 'teacher']
>>> e = dict(zip(x, y))
>>> print(e)
{'name': '陈丽', 'age': '18', 'job': 'teacher'}
```

#### 7.1.4 用 {}, dict 创建空字典

```
>>> f = {}
>>> print(f)
```

```
{}
```

---

```
>>> g = dict()
>>> print(g)
{}

```

### 7.1.5 用 fromkeys 创建‘值为空’的字典

```
>>> h = dict.fromkeys(['name', 'age', 'job'])
>>> print(h)
{'name': None, 'age': None, 'job': None}

```

## 7.2 字典中元素访问方法

### 7.2.1 通过“键”获取“值”。若“键”不存在，则抛出异常

```
>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> a['name']
'陈丽'
>>> a['age']
18
>>> a['job']
'teacher'

```

#若键不存在，抛出异常。

```
>>> a['names']
Traceback (most recent call last):
  File "<pyshell#197>", line 1, in <module>
    a['names']
KeyError: 'names'

```

### 7.2.2 get() 方法：强烈推荐的字典元素访问

**优点：**若‘键’不存在，返回的是 None, 而不是抛出异常。还可以设定，当‘键’不存在时，自己设定默认的返回对象。

```

>>> b = {'name': '陈丽', 'age': 18, 'job': ['teacher', 'wife', 'sister']}
>>> b.get('name')
'陈丽'
#若'键'不存在, 返回的是 None, 而不是抛出异常。
>>> b.get('sex')
>>> b.get('sex') == None
True
>>> if b.get('sex') == None:
    print('我爱你')
我爱你
#当'键'不存在时, 自己设定默认的返回对象。
>>> b.get('sex', '不存在')
'不存在'

```

### 7.2.3 用 items 获取 ‘所有的键值对’

```

>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> a.items()
dict_items([('name', '陈丽'), ('age', 18), ('job', 'teacher')])
>>> type(a.items())
<class 'dict_items'>
>>> for i in a.items():
    print(i)
('name', '陈丽')
('age', 18)
('job', 'teacher')

```

### 7.2.4 列出所有有 ‘键’ : keys, 列出所有有 ‘值’ : values

```

>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> a.keys()
dict_keys(['name', 'age', 'job'])

```



```
>>> a.values()
dict_values(['陈丽', 18, 'teacher'])
```

### 7.2.5 字典键值对的个数: len()

```
>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> len(a)
3
```

## 7.3 向字典中“添加元素”

7.3.1 “键”存在，则覆盖原有“键值对”。“键”不存在，新增键值对  
格式：字典['键']: '值'。

```
>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> a['job'] = 'sister'    #覆盖原有“键值对”
>>> print(a)
{'name': '陈丽', 'age': 18, 'job': 'sister'}

>>> a['height'] = 170      #新增“键值对”
>>> print(a)
{'name': '陈丽', 'age': 18, 'job': 'sister', 'height': 170}
```

### 7.3.2 使用 update 把 b 字典的所有“键值对”添加到 a 字典中

**格式：**a.update(b)，若果“键值对重复”，直接覆盖；否则，合并。

```
>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> a
{'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> b = dict([('job', 'Python'), ('weight', 75), ('height', 170)])
>>> b
{'job': 'Python', 'weight': 75, 'height': 170}
>>> a.update(b)
>>> a
{'name': '陈丽', 'age': 18, 'job': 'Python', 'weight': 75, 'height': 170}
```

## 7.4 “删除”字典中元素

### 7.4.1 del 方法:删除指定的“键值对”

**注:** 一旦删除‘键’和‘值’就消失了。

```
>>> a = {'name': '陈丽', 'age': 18, 'job': 'teacher'}
>>> del(a['job'])
>>> print(a)
{'name': '陈丽', 'age': 18}
```

### 7.4.2 clear 方法:删除字典中所有的‘键值对’

**注:** 删除所有的, 有点狠。

```
>>> b = dict([('job', 'Python'), ('weight', 75), ('height', 170)])
>>> b
{'job': 'Python', 'weight': 75, 'height': 170}
>>> b.clear()
>>> b
{}
```

### 7.4.3 pop 方法: 删除指定的‘键’

**优点:** “键”删除后, 可以用一个“变量”接收返回的“值”。

```
>>> c = {'name': '黄伟', 'age': 18, 'job': 'teacher'}
>>> cc = c.pop('name')
>>> cc
'黄伟'
>>> c
{'age': 18, 'job': 'teacher'}
```

### 7.4.4 popitem 方法:随机删除和返回‘键值对’

**注:** 字典是无序可变序列, 没有第一个元素、最后一个元素的概念

```
>>> b = {'name': '陈丽', 'age': 18, 'job': ['teacher', 'wife']}
>>> bb = b.popitem()
>>> print(bb)
('job', ['teacher', 'wife'])
>>> bbb = b.popitem()
```

```

>>> print(bbb)
('age', 18)
>>> bbbb = b.popitem()
>>> print(bbbb)
('name', '陈丽')
>>> bbbbbb = b.popitem()
>>> print(bbbbbb)
Traceback (most recent call last):
  File "<pyshell#296>", line 1, in <module>
    bbbbbb = b.popitem();print(bbbbbb)
KeyError: 'popitem(): dictionary is empty'

```

## 7.5 序列解包：运用于字典（类似于赋值）

### 7.5.1 利用 items：把‘键值对’赋给 b, c, d, e, f...

```

>>> a = {'name1': '陈丽', 'name2': '黄伟', 'name3': '阿亮', 'name4': '荣哥'}
>>> b, c, d, e = a.items()
>>> b, c
(('name1', '陈丽'), ('name2', '黄伟'))

```

### 7.5.2 利用 keys：把‘键’赋给 g, h, i, j, k...

```

>>> a = {'name1': '陈丽', 'name2': '黄伟', 'name3': '阿亮', 'name4': '荣哥'}
>>> g, h, i, j = a.keys()
>>> g, h
('name1', 'name2')

```

### 7.5.3 利用 values：把‘值’赋给 l, m, n, o, p...

```

>>> a = {'name1': '陈丽', 'name2': '黄伟', 'name3': '阿亮', 'name4': '荣哥'}
>>> l, m, n, o = a.values()
>>> l, m
('陈丽', '黄伟')

```

## 8 格式化输出：%s 和 format 的用法

### 8.1 python 格式化历史起源

- 1、python2.5 之前，我们使用的是老式格式化。
- 2、python3.0 开始（python2.6）同期发布，同时支持两个版本的格式化，出来一个新版本。
- 3、为什么要学习新式的 python3 格式化语法？

因为，虽然老式的语法，它兼容性很好，并且和大多数语言一样，但是它，功能很少，很难完成复杂的任务。

### 8.2 基本格式化(位置格式化)

```
新版

>>> '{qsfghnv}{fkbmth}{wjndv}{vogmh}'.format(6, 3, 4, 1)
'6qsfghnv3fkbmth4wjndv1vogmh'
>>> '{1}-{0}'.format(1, 2) # 新版格式化，支持带索引的顺序
'2-1'

旧版

>>> '%s %s %s' % (2, 3, 1)
'2 3 1'

提示：旧版格式化，不支持带索引的顺序。
```

### 8.3 填充和对齐

#### 8.3.1 填充

**概念：**当我们指定了字符串必须要有的长度的时候，如果现有的字符串没有那么长，那么我们就用某种字符（填充字符）来填满这个长度。填充以后，一定会有一个默认的对齐。

```
新版

>>> '{}'.format('left')
'left'
>>> '{:10}'.format('left')
```

```
'left      '
```

### 采用任意填充字符

```
>>> '{:_<10}'.format('left')
```

```
'left_____'
```

```
>>> '{:_>10}'.format('left')
```

```
'_____left'
```

```
>>> '{:_^10}'.format('left')
```

```
'__left____'
```

提示：不能直接在长度 10 面前加“填充符号”，因为无法区分。

### 旧版

```
>>> '%s' % ('right')
```

```
'right'
```

```
>>> '%10s' % ('right')
```

```
'      right'
```

## 8.3.2 对齐

**概念：**因为我们选择在某一端填充，会偏移到某一个方向。

### 新版：默认对齐是左对齐

```
>>> '{:<10}'.format('left') #左对齐
```

```
'left      '
```

```
>>> '{:>10}'.format('left') #右对齐
```

```
'      left'
```

```
>>> '{:^10}'.format('left') #居中对齐
```

```
'  left  '
```

### 旧版：默认对齐是右对齐

```
>>> '%10s' % ('right') #右对齐
```

```
'      right'
```

```
>>> '%-10s' % ('right')      #左对齐
'right'
```

**注：**旧版语法不支持居中对齐。同时，想要修改“填充符号”，很麻烦，不直接，因此不用学习。

### 8.3.3 截断

**概念：**如果我们指定的“截断长度”，比实际给出的“字符串”的长度要短，会发生截断。

1、使用的是长度而不是截断长度，如果实际长度超过了指定长度，那么指定长度无效。

```
新版
>>> '{:8}'.format('Huangwei is Beautiful')
'Huangwei is Beautiful'
```

2、使用的是截断长度，如果实际长度超过了指定长度，会发生截断。

```
新版
>>> '{:.8}'.format('Huangwei is Beautiful')
'Huangwei'

旧版
>>> '%.8s' % ('Huangwei is Beautiful')
'Huangwei'
```

### 8.3.4 填充和截断结合使用

```
新版
>>> '{:10.8}'.format('Huangwei is Beautiful')
'Huangwei  '
```

**注：**10 是总长度，.8 是截断长度。一般是截断以后，再用默认 2 个空格填充。

```
>>> '{:_>10.8}'.format('Huangwei is Beautiful')
'__Huangwei'
>>> '{:<10.8}'.format('Huangwei is Beautiful')
'Huangwei__'
>>> '{:_^10.8}'.format('Huangwei is Beautiful')
'_Huangwei_'
```

**注：**“填充字符”是在总长度左边写“> < ^”+“填充字符”。

### 旧版

```
>>> '%10.8s' % ('Huangwei is Beautiful')
'   Huangwei'
>>> '%-10.8s' % ('Huangwei is Beautiful')
'Huangwei   '
```

## 8.4 和数字相关的语法

### 8.4.1 填充整数，使用 d

```
>>> '{} , {}'.format(1,2)
'1,2'
```

**问题：**明明是字符串，为什么可以把数字填进去？

```
>>> '123'+456
'123456'
>>> '123'+456
Traceback (most recent call last):
  File "<pyshell#419>", line 1, in <module>
    '123'+456
TypeError: must be str, not int
```

那么'{} , {}'.format(1,2)是怎么做到的呢？

原来是 python 自动帮我们做了一次转换运算，但是做“隐式转换”是需要代价的，它会消耗额外的性能，牺牲速度。

如果确定是一个数字填充，应该用以下做法：

```
>>> '{:d}'.format(3000)          #d 是 digit(数字)的简称
'3000'
>>> '{:d}'.format('3000')
Traceback (most recent call last):
  File "<pyshell#423>", line 1, in <module>
    '{:d}'.format('3000')
ValueError: Unknown format code 'd' for object of type 'str'
```

**注意：**因为:d 已经指明了只能填充一个数字，所以当我们传入“字符串”，就会报错。

#### 8.4.2 填充浮点数，使用 f

这里又有一个问题，虽然:d 指明填充的是数字，但传入一个浮点数(float)时，仍然会报错，此时，应该怎么办呢？

```
>>> '{:d}'.format(3.1415926)
Traceback (most recent call last):
  File "<pyshell#429>", line 1, in <module>
    '{:d}'.format(3.1415926)
ValueError: Unknown format code 'd' for object of type 'float'

>>> '{:f}'.format(3.1415926)      #f 是 float(浮点数)的简称
'3.141593'
```

**注：**在这里，f 默认的精度是 6 位小数，这个问题源自于 C 语言(不用知道为什么)。但是会有以下解决方案。



### 8.4.3 数字填充问题

新版

```
>>> '{:10d}'.format(3000)
'      3000'
>>> '{:>10d}'.format(3000)    #右对齐
'      3000'
>>> '{:<10d}'.format(3000)    #左对齐
'3000      '
>>> '{:^10d}'.format(3000)    #居中对齐
'   3000   '
>>> '{:6.2f}'.format(3.141592653589793)
'   3.14'
>>> '{:.2f}'.format(3.141592653589793)
'3.14'
```

注意一下三个区别：

```
>>> '{:10f}'.format(3.141592653589793)
'   3.141593'
>>> '{:.10f}'.format(3.141592653589793)
'3.1415926536'
>>> '{:13.10f}'.format(3.141592653589793)
'   3.1415926536'
```

没有“.”这个10表示保留字符串的总长度为10位，有“.”以后，左边数字表示是保留的字符串的总长度，右边是保留小数点的位数。

注意以下很重要的问题：

```
>>> '{:6.4f}'.format(3.141592653589793)
'3.1416'
>>> '{:6.5f}'.format(3.141592653589793)
'3.14159'
>>> '{:6.6f}'.format(3.141592653589793)
'3.141593'
```

**注：**上面实际返回的数字保留的总位数大于我们设定的 6。因为，保留的小数为 4、5、6，但是前面的“3.”又不能丢，所以长度就失效了。

#### 8.4.4 修改填充符号

```
>>> '{:_<6.2f}'.format(3.141592653589793)
'3.14__'
>>> '{: _>6.2f}'.format(3.141592653589793)
'__3.14'
>>> '{:_^6.2f}'.format(3.141592653589793)
'_3.14_'
```

#### 8.4.5 数字的正负号问题

```
>>> '{:+d}'.format(3)
'+3'
>>> '{:+d}'.format(-3)
'-3'
```

#### 8.4.6 d 和 f 前面有一个空格作用：保持对齐

```
>>> '{:d}'.format(3)
'3'
>>> '{:d}'.format(-3)
'-3'

>>> '{: d}'.format(3)
' 3'
>>> '{: d}'.format(-3)
'-3'
```

#### 旧版

```
>>> '%d' % (3000)
'3000'
>>> '%f' % (3.1415926)
'3.141593'
>>> '%10f' % (3.1415926)
'      3.141593'

>>> '%.2f' % (3.1415926)
'3.14'
>>> '%5.2f' % (3.1415926)
' 3.14'
```

**注：**想要用老式语法填充其它东西，不方便，最好使用新式语法填充。

## 9 集合(Set)

### 9.1 集合的作用

- 1 **去重**：把一个列表变成集合，就自动去重了。
- 2 **关系测试**：测试两组数据之前的交集、差集、并集等关系。

### 9.2 集合的特征

- 1、集合使用 set 表示；
- 2、集合也使用 {} 表示，与字典不同的是：字典中存储的是键值对，集合中存储的是单一的元素；
- 3、**注意 1**：x = {} 表示的是空字典，不表示集合；
- 4、**注意 2**：x = set() 可以创建空集合；
- 5、集合中不含有重复元素，集合自动过滤重复元素；

```
>>> x = {1, 2, 3, 1, 2, 3}
>>> len(x)
3
>>> x
{1, 2, 3}
```

- 6、集合中的元素——**无序性**
- 7、集合中元素类型，必须是可哈希类型——**不懂**；  
一个对象在其生命周期内，如果保持不变，就是 hashable（可哈希的），像 tuple 和 string 是可哈希的，list、set 和 dictionary 都是不可哈希的。

```
>>> x = {[1, 2, 3], 4}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> y = [(1, 2, 3), 4]
>>> y
[(1, 2, 3), 4]
```

8、集合的底层，就是以字典中的 key 来实现的，集合中的元素就会成为字典的 key，然后绑定一个固定的值，因此，集合与字典具有很大的相似性；

在集合中：x = {1, 2, 3}

在字典中：x = {1:None, 2:None, 3:None}

## 9.3 集合中提供的方法

### 9.3.1 add(): 向集合中添加指定的元素

```
>>> s = {1, 2, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

注：重复元素没办法加入。

### 9.3.2 remove(): 删除集合中指定的元素。如果元素不存在，报错

```
>>> s = {1, 2, 3}
>>> s.remove(2)
>>> s
{1, 3}
>>> s.remove(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 12
```

### 9.3.3 discard(): 删除集合中指定的元素。如果元素不存在，不进行任何操作

```
>>> s = {1, 2, 3}
>>> s.discard(2)
>>> s
{1, 3}
>>> s.discard(12)
>>> s.discard(12) == None
True
```

#### 9.3.4 pop(): 删除并返回任意集合中元素——一般不用

```
>>> s = {1, 2, 3}
>>> s.pop()
1
>>> s.pop()
2
>>> s.pop()
3
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

#### 9.3.5 clear(): 删除集合中的所有元素

```
>>> s = {1, 2, 3}
>>> s.clear()
>>> s
set()
```

#### 9.3.6 copy(): 对集合进行复制。(深拷贝? 浅拷贝)

```
>>> s = {1, 2, 3}
>>> t = s.copy()
>>> t
{1, 2, 3}
>>> s is t
False
```

9.3.7 difference(): 两个集合的差集，产生新的集合，但不改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.difference(t)
{1}
>>> s
{1, 2, 3}
```

注：返回当前集合中存在，但参数集合中不存在的元素，以集合返回。

9.3.8 difference\_update(): 功能与 difference() 相同，但改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.difference_update(t)
>>> s
{1}
```

注：因为改变了 s 原有集合，所以 s.difference\_update(t) 返回的是 None，s 改变了，我们只需要操作 s 本身即可。

9.3.9 intersection(): 两个集合的交集，产生新的集合，但不改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.intersection(t)
{2, 3}
>>> s
{1, 2, 3}
```



9.3.10 `intersection_update()`: 功能与 `difference()` 相同, 但改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.intersection_update(t)
>>> s
{2, 3}
```

9.3.11 `union()`: 两个集合的并集, 产生新的集合, 但不改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.union(t)
{1, 2, 3, 4}
>>> s
{1, 2, 3}
```

9.3.12 `update()`: 功能与 `union()` 相同, 但改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.update(t)
>>> s
{1, 2, 3, 4}
>>> t
{2, 3, 4}
```

### 9.3.13 symmetric\_difference(): 两个集合的对称差集

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.symmetric_difference(t)
{1, 4}
>>> s
{1, 2, 3}
```

注：“对称差集”返回 s 中有，t 中有，但不同时在 s、t 中共有的元素。

### 9.3.14 symmetric\_difference\_update(): 功能与 symmetric\_difference () 相同，但改变当前集合

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> s.symmetric_difference_update(t)
>>> s
{1, 4}
```

### 9.3.15 isdisjoint(): 判断当前集合与参数集合，是否交集为空；是返回 True，否返回 False

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> u = {4, 5, 6}
>>> s.isdisjoint(t)
False
>>> s.isdisjoint(u)
True
>>> s
{1, 2, 3}
```

9.3.16 `issubset()`: 判断当前集合是否为参数集合, 的子集; 是返回 `True`, 否返回 `False`

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> u = {4, 5, 6}
>>> s.issubset(t)
False
>>> s.issubset(u)
False
>>> s
{1, 2, 3}
```

9.3.17 `issuperset()`: 判断当前集合是否为参数集合, 的父集; 是返回 `True`, 否返回 `False`

```
>>> s = {1, 2, 3}
>>> t = {2, 3, 4}
>>> u = {1, 2, 3, 4}
>>> v = {2, 3}
>>> s.issuperset(t)
False
>>> s.issuperset(v)
True
>>> s
{1, 2, 3}
```

## 9.4 集合的运算

### 9.4.1 **in**: 判断某个元素是否在集合中，是返回 True，否返回 False

```
>>> s = {1, 2, 3, 4}
>>> 2 in s
True
>>> 6 in s
False
```

### 9.4.2 **&**: 返回两个元素的交集，相当于 `s.intersection(t)`

```
>>> s = {1, 2, 3, 4, 5}
>>> t = {4, 5, 6, 7}
>>> s & t
{4, 5}
>>> s
{1, 2, 3, 4, 5}
```

### 9.4.4 **|**: 返回两个元素的并集，相当于 `s.union(t)`

```
>>> s = {1, 2, 3, 4, 5}
>>> t = {4, 5, 6, 7}
>>> s | t
{1, 2, 3, 4, 5, 6, 7}
```

### 9.4.5 **-**: 返回两个元素的差集，相当于 `s.difference(t)`

```
>>> s = {1, 2, 3, 4, 5}
>>> t = {4, 5, 6, 7}
>>> s - t
{1, 2, 3}
```

9.4.6  $\wedge$ : 返回两个元素的对称差集, 相当于 `s.symmetric_difference(t)`

```
>>> s = {1, 2, 3, 4, 5}
>>> t = {4, 5, 6, 7}
>>> s ^ t
{1, 2, 3, 6, 7}
```

## 9.5 集合比较运算

9.5.1 `==`: 两个集合中元素是否一致

```
>>> s = {1, 2, 3, 4, 5}
>>> t = {4, 5, 6, 7}
>>> u = {1, 2, 3, 4, 5}
>>> s == t
False
>>> s == u
True
```

9.5.2 `>`、`>=`、`<`、`<=`: 父集、子集比较

```
>>> s = {1, 2, 3, 4, 5}
>>> t = {4, 5, 6, 7}
>>> u = {1, 2, 3}
>>> s > t
False
>>> s > u
True
>>> u < t
False
>>> u < s
True
```

**注:** 若  $A > B$ , 则  $A$  是  $B$  的父集,  $A$  是  $B$  的子集;

### 9.5.3 is、is not: 比较两个集合是否是同一个对象

```
>>> s = {1, 2, 3}
>>> t = s.copy()
>>> t
{1, 2, 3}
>>> s == t
True
>>> s is t
False
```

### 9.6 集合推导式: 可以自动去重

```
>>> s = {i for i in range(10)}
>>> s
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> t = {i for i in [1, 2, 3, 4, 2, 3]}
>>> t
{1, 2, 3, 4}
```

## 9.7 真值测试

```
>>> #整数int类型
>>> x = 10
>>> y = 0
>>> bool(x), bool(y)
(True, False)

>>> #浮点数float类型
>>> x = 2.5
>>> y = 0.0
>>> bool(x), bool(y)
(True, False)

>>> #复数complex类型
>>> x = 3 + 4j
>>> y = 0j
>>> bool(x), bool(y)
(True, False)

>>> #序列(list, tuple, str, bytes)类型
>>> bool([]), bool([1, 2, 3, 2, 3])
(False, True)
>>> bool(()), bool((1, 2, 3, 2, 3))
(False, True)
>>> bool(""), bool("Huang Wei")
(False, True)
>>> bool(b""), bool(b"Chen Li")
(False, True)

>>> #集合类型
>>> bool(set()), bool({1, 2, 3, 2, 3})
(False, True)
>>> #None会转换为False
>>> bool(None)
False
```

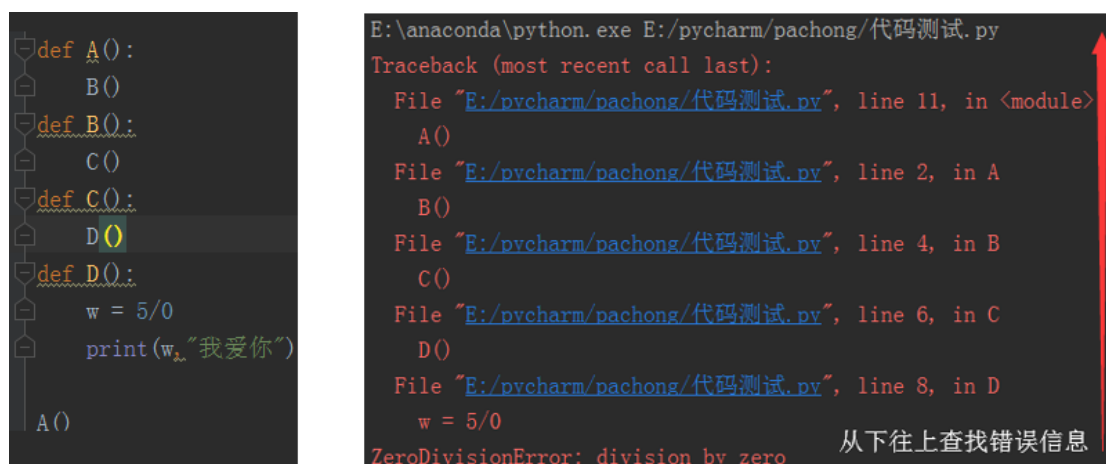
## 10 异常

### 10.1 异常的概念

异常是一种错误，是程序在运行过程中，产生的突发事件，该事件会干扰程序的正常流程。当异常产生时，如果没有对异常进行处理，则异常后的代码不会得到执行。

比如说在 Pycharm 中，若当前模块是作为脚本运行的，则异常会插播给解释器，如果到了解释器，异常还未解决，则当前线程会终止运行。此时会在控制台打印错误信息。

注意：离着异常越近的信息，越在下面显示。因此，我们分析异常时，应该从下向上进行查看。



The image shows a PyCharm IDE window with a Python script on the left and a traceback in the console on the right. The script defines four functions: A(), B(), C(), and D(). Function A() calls B(), B() calls C(), C() calls D(), and D() contains a division by zero error (w = 5/0) and a print statement. The console shows the traceback starting from the module level, through A(), B(), C(), and finally D() where the ZeroDivisionError occurred. A red arrow on the right points upwards, indicating that the error message at the bottom is the most recent and relevant information.

```
def A():  
    B()  
def B():  
    C()  
def C():  
    D()  
def D():  
    w = 5/0  
    print(w, "我爱你")  
A()
```

```
E:\anaconda\python.exe E:/pycharm/pachong/代码测试.py  
Traceback (most recent call last):  
  File "E:/pycharm/pachong/代码测试.py", line 11, in <module>  
    A()  
  File "E:/pycharm/pachong/代码测试.py", line 2, in A  
    B()  
  File "E:/pycharm/pachong/代码测试.py", line 4, in B  
    C()  
  File "E:/pycharm/pachong/代码测试.py", line 6, in C  
    D()  
  File "E:/pycharm/pachong/代码测试.py", line 8, in D  
    w = 5/0  
ZeroDivisionError: division by zero
```

从下往上查找错误信息



## 10.2 常见异常类型

```
>>> # BaseException Python所有异常类的根类
>>> # Exception 是BaseException的子类。用户异常的根类，我们自定义的异常应该继承
    该类
>>> # ZeroDivisionError 当0做除数时，会产生该异常
>>> 5/0
Traceback (most recent call last):
  File "<pyshell#214>", line 1, in <module>
    5/0
ZeroDivisionError: division by zero
>>>
>>> # AssertionError 断言异常。当断言条件不满足时，会产生该异常
>>> assert 5 > 10
Traceback (most recent call last):
  File "<pyshell#217>", line 1, in <module>
    assert 5 > 10
AssertionError
>>>
>>> # ModuleNotFoundError 当使用import导入的模块不存在时，会产生该异常
>>> import aaa
Traceback (most recent call last):
  File "<pyshell#220>", line 1, in <module>
    import aaa
ModuleNotFoundError: No module named 'aaa'
>>>
>>> # IndexError 索引异常。当索引越界时，会产生该异常
>>> li = [1, 2]
>>> li[5]
Traceback (most recent call last):
  File "<pyshell#224>", line 1, in <module>
    li[5]
IndexError: list index out of range
```

```
>>> # KeyError 键访问异常。当字典中，键不存在时，会产生该异常
>>> dic = {"a":1, "b":2}
>>> dic["c"]
Traceback (most recent call last):
  File "<pyshell#228>", line 1, in <module>
    dic["c"]
KeyError: 'c'
>>>
>>> # NameError 名称异常。当访问的变量不存在或没定义时，会产生该异常
>>> xxx
Traceback (most recent call last):
  File "<pyshell#231>", line 1, in <module>
    xxx
NameError: name 'xxx' is not defined
>>>
>>> # RecursionError 递归异常。当函数调用层次达到了最大层次限制时(通常但不绝对是
    递归)，会产生该异常
>>> def x():
>>>     x()
>>>
>>> x()
Traceback (most recent call last):
  File "<pyshell#237>", line 1, in <module>
    x()
  File "<pyshell#236>", line 2, in x
    x()
  File "<pyshell#236>", line 2, in x
    x()
  File "<pyshell#236>", line 2, in x
    x()
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
```

```

>>> # StopIteration 终止迭代异常。当迭代器没有元素时，会产生该异常
>>> li = [1]
>>> it = li.__iter__()
>>> it.__next__()
1
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#243>", line 1, in <module>
    it.__next__()
StopIteration
>>>
>>> # SyntaxError 语法错误异常
>>> for i in range(6)
SyntaxError: invalid syntax
>>>
>>> # ValueError 值异常。当对值处理错误时，会产生该异常
>>> int("abc")
Traceback (most recent call last):
  File "<pyshell#249>", line 1, in <module>
    int("abc")
ValueError: invalid literal for int() with base 10: 'abc'

```

至此，Python 基础所有内容已经更新完毕，如果你好好看的话，我相信你一定会有很大的收获。