

1. Python 2 与 3 的区别

- 1.1. `print` 函数
- 1.2. `Unicode` 编码
- 1.3. 除法运算
- 1.4. 异常与抛出
- 1.5. `range()` 取代 `xrange()`
- 1.6. 进制数表示
- 1.7. 不等运算符
- 1.8. 去掉了 `repr` 表达式``
- 1.9. 模块更名
- 1.10. 数据类型
- 1.11. 打开文件及输入
- 1.12. `map`、`filter` 和 `reduce`

2. Python 解释器

- 2.1. 环境变量
- 2.2. 脚本式编程

3. Python3 基础语法

- 3.1. 编码
- 3.2. 标识符
- 3.3. 保留字
- 3.4. 注释
- 3.5. 行与缩进
 - 3.5.1. 缩进
 - 3.5.2. 空行
- 3.6. 多行语句
 - 3.6.1. 一条语句跨多行
 - 3.6.2. 一行写多条语句
 - 3.6.3. 多个语句构成代码组
- 3.7. 输入与输出
 - 3.7.1. `input` 输入
 - 3.7.2. `print` 输出
- 3.8. `import` 与 `from ... import`
- 3.9. `help()` 函数
- 3.10. 命令行参数
- 3.11. `if __name__ == '__main__':` 的作用

4. Python 3 基本数据类型

- 4.1. 变量赋值
- 4.2. 标准数据类型
- 4.3. 查询对象类型
- 4.4. 数字 (Number)
 - 4.4.1. 数字类型
 - 4.4.2. 数值运算
 - 4.4.3. 数学函数
 - 4.4.4. 随机数函数
 - 4.4.5. 三角函数

- 4. 4. 6. 数学常量
- 4. 4. 7. 其他说明
- 4. 5. 字符串 (String)
 - 4. 5. 1. 字符串的截取
 - 4. 5. 2. 字符串的操作
 - 4. 5. 3. 字符串的格式化
 - 4. 5. 3. 1. 使用 "%" 操作符
 - 4. 5. 3. 2. 使用 `str.format()`
 - 4. 5. 3. 3. 使用 `f-string` 来格式化
 - 4. 5. 4. 字符串的转义
 - 4. 5. 5. 字符串的内建函数
- 4. 6. 列表 (List)
 - 4. 6. 1. 列表的截取
 - 4. 6. 2. 列表的操作
 - 4. 6. 3. 列表的函数与方法
- 4. 7. 元组 (Tuple)
 - 4. 7. 1. 元组的截取
 - 4. 7. 2. 元组的操作
 - 4. 7. 3. 元组的内置函数
- 4. 8. 集合 (Set)
- 4. 9. 字典 (Dictionary)
- 4. 10. Python 数据类型转换
- 4. 11. 数组,列表,矩阵之间的相互转化

5. Python 3 数据结构

- 5. 1. 列表作堆栈
- 5. 2. 列表推导式
- 5. 3. 遍历技巧

6. Python 3 运算符

- 6. 1. 算术运算符
- 6. 2. 比较(关系)运算符
- 6. 3. 赋值运算符
- 6. 4. 逻辑运算符
- 6. 5. 位运算符
- 6. 6. 成员运算符
- 6. 7. 身份运算符
- 6. 8. 运算符优先级
- 6. 9. 无自增/自减运算

7. Python 3 控制语句

- 7. 1. 条件控制
 - 7. 1. 1. `if` 语句
 - 7. 1. 2. `if` 嵌套
- 7. 2. 循环控制
 - 7. 2. 1. `while` 循环
 - 7. 2. 2. `for` 循环
 - 7. 2. 3. `pass`, `break` 和 `continue` 语句

8. Python 3 迭代器与生成器

- 8. 1. 迭代器
 - 8. 1. 1. 迭代器的创建
 - 8. 1. 2. 类作为迭代器

8.2. 生成器

9. Python 3 函数

9.1. 定义函数

9.2. 函数调用

9.3. 参数传递

9.4. 参数类型

9.5. 匿名函数 (`lambda`)

9.6. 强制位置参数

10. Python 3 变量前加 * 或 ** 号

10.1. 变量前加 * 号可进行拆分

10.2. 函数传参中使用 * 或 **

10.3. 综合以上两点的实例

10.4. 使用 `zip()` 函数进行压缩

11. Python 3 命名空间和作用域

12. Python 3 模块

13. Python 3 输入和输出

14. Python 3 文件操作

15. Python 3 OS操作

16. Python 3 错误和异常

17. Python 中的浅拷贝与深拷贝

17.1. 赋值语句

17.2. 浅拷贝

17.3. 深拷贝

18. Python 获取命令行参数

18.1. 利用 `sys.argv`

18.1.1. 实例1

18.1.2. 实例2

18.2. 利用 `getopt` 模块

18.2.1. `getopt.getopt` 方法

18.2.2. `getopt.gnu_getopt` 方法

18.2.3. 异常处理 `except getopt.GetoptError`

18.2.4. 实例

Python3 基础教程

1. Python 2 与 3 的区别

1.1. `print` 函数

`print` 语句没有了，取而代之的是 `print()` 函数。

在 Python 2.6 与 Python 2.7 里面，以下三种形式是等价的：

```
1 print "fish"
2 print ("fish") # 注意print后面有个空格
3 print("fish") # print()不能带有任何其它参数
```

然而，Python 2.6 实际已经支持新的 `print()` 函数语法：

```
1 from __future__ import print_function
2 print("fish", "panda", sep=',')
```

1.2. Unicode 编码

Python 2 有 ASCII `str()` 类型，`unicode()` 是单独的，不是 `byte` 类型。现在在 Python 3 中，我们有了 `Unicode (utf-8)` 字符串，以及一个字节类：`byte` 和 `bytearrays`。由于 Python3.X 源码文件默认使用 `utf-8` 编码，这就使得以下代码是合法的：

```
1 >>> 中国 = 'china'
2 >>> print(中国)
3 china
```

1.3. 除法运算

在 Python 3 中对于整数之间的相除 (`/`)，结果也会是浮点数。

而对于 `//` 除法 (floor除法)，会自动对结果进行一个 `floor` 操作，这在 Python 2 和 3 中是一致的。

1.4. 异常与抛出

Python 3 中使用 `as` 作为关键词。捕获异常的语法由 `except exc, var` 改为 `except exc as var`。使用语法 `except (exc1, exc2) as var` 可以同时捕获多种类别的异常。Python 2.6 已经支持这两种语法。

此外：

1. 在 Python 2.x 时代，所有类型的对象都是可以直接被抛出的；而在 Python 3.x 时代，只有继承自 `BaseException` 的对象才可以被抛出。
2. 在 Python 2.x 中 `raise` 语句使用逗号将抛出对象类型和参数分开；而 Python 3.x 中取消了这种奇葩的写法，直接调用构造函数抛出对象即可。

1.5. `range()` 取代 `xrange()`

在 Python 3 中，`range()` 是像 `xrange()` 那样实现的，以至于一个专门的 `xrange()` 函数不再存在。在 Python 3 中使用 `xrange()` 会抛出命名异常。

1.6. 进制数表示

1. 八进制数必须写成：`0o777`，原来的形式：`0777` 不能用了；二进制必须写成：`0b111`。
2. 新增了一个 `bin()` 函数用于将一个整数转换成二进制字符串。

1.7. 不等运算符

Python 3 中去掉了 `<>`，只有 `!=` 一种写法。

1.8. 去掉了 repr 表达式``

Python 2 中反引号``相当于 repr 函数的作用；
Python 3 中去掉了这种写法，只允许使用 repr() 函数。

1.9. 模块更名

Old Name	New Name	Old Name	New Name
<code>_winreg</code>	<code>winreg</code>	<code>ConfigParser</code>	<code>configparser</code>
<code>copy_reg</code>	<code>copyreg</code>	<code>Queue</code>	<code>queue</code>
<code>SocketServer</code>	<code>socketserver</code>	<code>repr</code>	<code>reprlib</code>

StringIO 被合并到 io 模组内；new, md5, gopherlib 等模块被删除；httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib 被合并到 http 包内；取消了 exec 语句，只剩下 exec() 函数。

1.10. 数据类型

1. Python 3 去除了 long 类型，现在只有一种整型 int，但它的行为就像 Python 2 版本的 long。
2. 新增了 bytes 类型，对应于 Python 2 版本中的八位串，定义一个 bytes 变量的方法如下：

```
1 >>> b = b'china'
2 >>> type(b)
3 <type 'bytes'>
```

str 对象和 bytes 对象可以使用 .encode() (str -> bytes) 或 .decode() (bytes -> str) 相互转化。

```
1 >>> s = b.decode()
2 >>> s
3 'china'
4 >>> b1 = s.encode()
5 >>> b1
6 b'china'
```

3. dict 类型的 .keys()、.items() 和 .values() 方法返回迭代器，而之前的 .iterkeys() 等函数都被废弃。同时去掉的还有 dict.has_key()，用 in 替代它吧。

1.11. 打开文件及输入

1. 原 Python 2 中：

```
1 file( ..... )
2 或
3 open(.....)
```

现改为只能用：

```
1 | open(.....)
```

2. Python 3 中 `input()` 函数替代了原 `raw_input()` 函数，其接收任意性输入，将所有输入默认为字符串处理，并返回字符串类型。

1.12. `map`、`filter` 和 `reduce`

这三个函数号称是函数式编程的代表。

在 Python 2 中，它们都是内置函数 (built-in function)。

在 Python 3 中，它们从内置函数变成了类 (class)；其次它们的返回结果也从当初的列表变成了一个可迭代的对象，可以使用 `next()` 函数来进行手工迭代。

2. Python 解释器

2.1. 环境变量

Variable	Description
<code>PYTHONPATH</code>	Python搜索路径，默认 <code>import</code> 的模块都会从 <code>PYTHONPATH</code> 中寻找
<code>PYTHONSTARTUP</code>	Python启动后，先执行 <code>PYTHONSTARTUP</code> 环境变量指定的文件中的代码
<code>PYTHONCASEOK</code>	加入 <code>PYTHONCASEOK</code> 的环境变量，会使Python导入模块时不区分大小写
<code>PYTHONHOME</code>	模块搜索路径，通常内嵌于 <code>PYTHONSTARTUP</code> 或 <code>PYTHONPATH</code> 目录中，使得模块库更容易切换

2.2. 脚本式编程

在Linux/Unix系统中，你可以在脚本顶部添加以下命令让Python脚本可以像SHELL脚本一样可直接执行：

```
1 | #!/usr/bin/env python3
```

然后修改脚本权限，使其有执行权限：

```
1 | $ chmod +x hello.py
```

执行以下命令即可直接运行脚本：

```
1 | ./hello.py
```

3. Python3 基础语法

3.1. 编码

默认情况下，Python 3 源码文件以 UTF-8 编码，所有字符串都是 Unicode 字符串，即：

```
1 | #_*_ coding:utf-8 *_
```

当然你也可以为源码文件指定不同的编码：

```
1 | # -*- coding: cp-1252 -*-
```

上述定义允许在源文件中使用 Windows-1252 字符集中的字符编码，对应适合语言为保加利亚语、白罗斯语、马其顿语、俄语、塞尔维亚语。

3.2. 标识符

- 1 | 第一个字符必须是字母表中字母(a-z,A-Z)或下划线(_)。
- 2 | 标识符的其他部分由字母、数字和下划线组成。
- 3 | 标识符对大小写敏感。

3.3. 保留字

保留字即关键字，我们不能把它们用作任何标识符名称。

Python 的标准库提供了一个 `keyword` 模块，可输出当前版本的所有关键字：

```
1 | >>> import keyword
2 | >>> keyword.kwlist
3 | ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
   'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from',
   'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
   'raise', 'return', 'try', 'while', 'with', 'yield']
```

3.4. 注释

单行注释：以 `#` 开头；

多行注释：多个 `#`，或者 `'''`，或者 `"""`。

```
1 | #!/usr/bin/env python3
2 | # 第一个注释
3 | # 第二个注释
4 | '''
5 | 第三注释
6 | 第四注释
7 | '''
8 | """
9 | 第五注释
10 | 第六注释
11 | """
12 | print ("Hello, Python!")
```

以下实例可以输出函数的注释：

```
1 | def a():
2 |     '''这是文档字符串'''
3 |     pass
4 | print(a.__doc__)
```

输出结果为：

```
1 | 这是文档字符串
```

3.5. 行与缩进

3.5.1. 缩进

Python最具特色的就是使用缩进来表示代码块，不需要使用大括号 {}。

缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

3.5.2. 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。

类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

3.6. 多行语句

Python 通常是一行写完一条语句。

3.6.1. 一条语句跨多行

但如果语句很长，我们可以使用反斜杠(\)来实现多行语句，例如：

```
1 | total = item_one + \  
2 |         item_two + \  
3 |         item_three
```

在 [], {}、或 () 中的多行语句，不需要使用反斜杠(\)，例如：

```
1 | total = ['item_one', 'item_two', 'item_three',  
2 |         'item_four', 'item_five']
```

3.6.2. 一行写多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，例如：

```
1 | #!/usr/bin/env python3  
2 | import sys; x = 'runoob'; sys.stdout.write(x + '\n')
```

使用脚本执行以上代码，输出结果为：

```
1 | runoob
```

使用交互式命令行执行，输出结果为：

```
1 | >>> import sys; x = 'runoob'; sys.stdout.write(x + '\n')  
2 | runoob  
3 | 7
```


此处的 7 表示字符数。

3.6.3. 多个语句构成代码组

缩进相同的一组语句构成一个代码块，称之代码组。像 `if`、`while`、`def` 和 `class` 这样的复合语句，首行以关键字开始并以冒号(:)结束，该行之后的一行或多行代码构成代码组。首行及后面的代码组称为一个子句 (clause)。

如下实例：

```
1  if expression1 :
2      suite1
3  elif expression2 :
4      suite2
5  else :
6      suite3
```

3.7. 输入与输出

3.7.1. `input` 输入

Python 3 仅保留了 `input()` 函数，它可接收任意任性输入，将所有输入默认为字符串处理，并返回字符串类型。

执行下面的程序在按回车键后就会等待用户输入：

```
1  #!/usr/bin/env python3
2  input("\n\n按下 enter 键后退出。")
```

以上代码中，`'\n\n'` 在结果输出前会输出两个新的空行。一旦用户按下 `Enter` 键时，程序将退出。

3.7.2. `print` 输出

`print` 默认输出是换行的 (即 `end='\n'`)，若要实现不换行需在变量末尾加上 `end=' '`，实例：

```
1  #!/usr/bin/env python3
2  x="a"
3  y="b"
4  # 换行输出
5  print(x)
6  print(y)
7  print('-----')
8  # 不换行输出
9  print(x, end=' ')
10 print(y, end=' ')
11 print()
```

以上实例执行结果为：

```
1  a b
2  -----
3  a b
```

通过命令 `help(print)` 我们知道这个方法里第二个为缺省参数 `sep=' '`，表示使用空格作为分隔符。

```

1 >>> help(print)
2 Help on built-in function print in module builtins:
3
4 print(...)
5     print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
6
7     Prints the values to a stream, or to sys.stdout by default.
8     Optional keyword arguments:
9     file: a file-like object (stream); defaults to the current sys.stdout.
10    sep:   string inserted between values, default a space.
11    end:   string appended after the last value, default a newline.
12    flush: whether to forcibly flush the stream.

```

所以在打印 `dict` 类的使用, 可以这样写 (使用冒号作为分隔符):

```

1 >>> def getPairs(dict):
2 ...     for k,v in dict.items() :
3 ...         print(k,v,sep=':')
4 ...
5 >>> getPairs({x:x**3 for x in (1,2,3,4)})

```

输出结果 :

```

1 1:1
2 2:8
3 3:27
4 4:64

```

3.8. `import` 与 `from ... import`

在 Python 用 `import` 或者 `from ... import` 来导入相应的模块。

Description

将整个模块(somemodule)导入 : `import somemodule`

从某个模块中导入某个函数 : `from somemodule import somefunction`

从某个模块中导入多个函数 : `from somemodule import firstfunc, secondfunc, thirdfunc`

将某个模块中的全部函数导入 : `from somemodule import *`

3.9. `help()` 函数

调用 Python 的 `help()` 函数可以打印输出一个函数的文档字符串, 按下 `:q` 即退出说明文档。

```

1  >>> help(max) # 查看 max 内置函数的参数列表和规范的文档
2  Help on built-in function max in module builtins:
3
4  max(...)
5      max(iterable, *, default=obj, key=func) -> value
6      max(arg1, arg2, *args, *, key=func) -> value
7
8      With a single iterable argument, return its biggest item. The
9      default keyword-only argument specifies an object to return if
10     the provided iterable is empty.
11     With two or more arguments, return the largest argument.
12 (END)

```

若仅想得到文档字符串：

```

1  >>> print(max.__doc__) # 注意，doc的前后分别是两个下划线
2  max(iterable, *, default=obj, key=func) -> value
3  max(arg1, arg2, *args, *, key=func) -> value
4
5  With a single iterable argument, return its biggest item. The
6  default keyword-only argument specifies an object to return if
7  the provided iterable is empty.
8  With two or more arguments, return the largest argument.

```

在 `print()` 打印的时候双引号与单引号都可作为定界符使用，且可以嵌套，被嵌套的会被解释为标点符号。

3.10. 命令行参数

很多程序可以执行一些操作来查看一些基本信息，Python可以使用 `-h` 参数查看各参数帮助信息：

```

1  $ python -h
2  usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
3  Options and arguments (and corresponding environment variables):
4  -b      : issue warnings about str(bytes_instance), str(bytearray_instance)
5           and comparing bytes/bytearray with str. (-bb: issue errors)
6  -B      : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
7  -c cmd  : program passed in as string (terminates option list)
8  -d      : debug output from parser; also PYTHONDEBUG=x
9  -E      : ignore PYTHON* environment variables (such as PYTHONPATH)
10 -h      : print this help message and exit (also --help)
11 -i      : inspect interactively after running script; forces a prompt even
12           if stdin does not appear to be a terminal; also PYTHONINSPECT=x
13 -I      : isolate Python from the user's environment (implies -E and -s)
14 -m mod  : run library module as a script (terminates option list)
15 -O      : remove assert and __debug__-dependent statements; add .opt-1 before
16           .pyc extension; also PYTHONOPTIMIZE=x
17 -OO     : do -O changes and also discard docstrings; add .opt-2 before
18           .pyc extension
19 -q      : don't print version and copyright messages on interactive startup
20 -s      : don't add user site directory to sys.path; also PYTHONNOUSERSITE
21 -S      : don't imply 'import site' on initialization
22 -u      : force the stdout and stderr streams to be unbuffered;
23           this option has no effect on stdin; also PYTHONUNBUFFERED=x
24 -v      : verbose (trace import statements); also PYTHONVERBOSE=x

```

```

25         can be supplied multiple times to increase verbosity
26 -V      : print the Python version number and exit (also --version)
27         when given twice, print more information about the build
28 -W arg  : warning control; arg is action:message:category:module:lineno
29         also PYTHONWARNINGS=arg
30 -x      : skip first line of source, allowing use of non-Unix forms of #!cmd
31 -X opt  : set implementation-specific option
32 --check-hash-based-pycs always|default|never:
33         control how Python invalidates hash-based .pyc files
34 file    : program read from script file
35 -        : program read from stdin (default; interactive mode if a tty)
36 arg ... : arguments passed to program in sys.argv[1:]
37
38 Other environment variables:
39 PYTHONSTARTUP: file executed on interactive startup (no default)
40 PYTHONPATH   : ':'-separated list of directories prefixed to the
41               default module search path. The result is sys.path.
42 PYTHONHOME   : alternate <prefix> directory (or <prefix>:<exec_prefix>).
43               The default module search path uses <prefix>/lib/pythonX.X.
44 PYTHONCASEOK : ignore case in 'import' statements (Windows).
45 PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
46 PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
47 PYTHONHASHSEED: if this variable is set to 'random', a random value is used
48               to seed the hashes of str and bytes objects. It can also be set to an
49               integer in the range [0,4294967295] to get hash values with a
50               predictable seed.
51 PYTHONMALLOC: set the Python memory allocators and/or install debug hooks
52               on Python memory allocators. Use PYTHONMALLOC=debug to install debug
53               hooks.
54 PYTHONCOERCECLOCALE: if this variable is set to 0, it disables the locale
55               coercion behavior. Use PYTHONCOERCECLOCALE=warn to request display of
56               locale coercion and locale compatibility warnings on stderr.
57 PYTHONBREAKPOINT: if this variable is set to 0, it disables the default
58               debugger. It can be set to the callable of your debugger of choice.
59 PYTHONDEVMODE: enable the development mode.
60 PYTHONPYCACHEPREFIX: root directory for bytecode cache (pyc) files.

```

3.11. `if __name__ == '__main__':` 的作用

一个 python 文件通常有两种使用方法:

第一是作为脚本直接执行;

第二是 `import` 到其他的 python 脚本中被调用 (模块重用) 执行。

`if __name__ == '__main__':` 的作用就是控制这两种情况执行代码的过程。在 `if __name__ == '__main__':` 下的代码只在第一种情况下 (即文件作为脚本直接执行时) 才会被执行, 而 `import` 到其他脚本中是不会被执行的。

4. Python 3 基本数据类型

Python 中的变量不需要声明。每个变量在使用前都必须赋值, 变量赋值以后该变量才会被创建。Python 中的变量就是变量, 它没有类型, 我们所说的"类型"是变量所指的内存中对象的类型。

4.1. 变量赋值

Python 中用等号 (=) 来给变量赋值, 等号运算符左边是一个变量名, 右边是存储在变量中的值。

```
1 | a = b = c = 1
```

Python 允许同时为多个变量赋值，例如：

```
1 | a, b, c = 1, 2, "runoob"
```

4.2. 标准数据类型

Python 3 中有六个标准的数据类型：

1. **Number** (数字)
2. **String** (字符串)
3. **List** (列表)
4. **Tuple** (元组)
5. **Set** (集合)
6. **Dictionary** (字典)

Python 3 的六个标准数据类型中：

- **不可变数据类型** (3个)： **Number** (数字)、 **String** (字符串)、 **Tuple** (元组)
- **可变数据类型** (3个)： **List** (列表)、 **Dictionary** (字典)、 **Set** (集合)

4.3. 查询对象类型

内置的 `type()` 函数可以用来查询变量所指的对象类型：

```
1 | >>> a, b, c, d = 20, 5.5, True, 4+3j
2 | >>> print(type(a), type(b), type(c), type(d))
3 | <class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

此外还可以用 `isinstance()` 来判断：

```
1 | >>> a = 111
2 | >>> isinstance(a, int)
3 | True
```

比较 `isinstance()` 和 `type()` 的区别在于：

```
1 | type()不会认为子类是一种父类类型；
2 | isinstance()会认为子类是一种父类类型。
3 |
4 | type()主要用于判断未知数据类型；
5 | isinstance()主要用于判断A类是否继承于B类。
```

实例：

```

1  # 判断子类对象是否继承于父类
2  class father(object):
3      pass
4
5  class son(father):
6      pass
7
8  if __name__ == '__main__':
9      print (type(son())==father)
10     print (isinstance(son(),father))
11     print (type(son()))
12     print (type(son))

```

运行结果：

```

1  False
2  True
3  <class '__main__.son'>
4  <type 'type'>

```

4.4. 数字 (Number)

4.4.1. 数字类型

Python中数字有四种类型：**整数 (int)**、**布尔型 (bool)**、**浮点数 (float)** 和**复数 (complex)**。

Number Type	Description/Example
<code>int</code> (整数)	如: 1。只有一种整数类型 <code>int</code> ，表示为长整型，没有python2中的 <code>long</code> 。
<code>bool</code> (布尔型)	如：True 或 False。其实它们的值还是1和0，可以和数字相加。
<code>float</code> (浮点数)	如：1.23、3.1E-2。
<code>complex</code> (复数)	如：1 + 2j、1.1 - 2.2j，a + bj 或 <code>complex(a, b)</code>

当你指定一个值时，**Number**对象就会被创建。

可以使用 `del` 语句删除一些对象引用：

```

1  del var1[,var2[,var3[...[,varN]]]]

```

注意：

1. 其他类型值转换为 `bool` 值时，除了 `' '`、`''`、`''''''`、`''''''''`、`0`、`()`、`[]`、`{}`、`None`、`0.0`、`0L`、`0.0+0.0j` 及 `False` 转换为 `False` 外，其他都为 `True`。
2. 虚数不能单独存在，它们总是和一个值为0.0的实数部分一起构成一个复数。获取复数 `x` 的实部 `x.real` 与虚部 `x.imag`；获取复数 `x` 的共轭: `x.conjugate()`。
3. Python 不支持复数转换为整数或浮点数。

4.4.2. 数值运算

```

1  >>> 5 + 4    # 加法
2  9

```

```
3 >>> 4.3 - 2 # 减法
4 2.3
5 >>> 3 * 7 # 乘法
6 21
7 >>> 2 / 4 # 除法，得到一个浮点数
8 0.5
9 >>> 2 // 4 # 除法，得到一个整数
10 0
11 >>> 17 % 3 # 取余
12 2
13 >>> 2 ** 5 # 乘方
14 32
```

4.4.3. 数学函数

Function	Description
<code>abs(x)</code>	返回一个数字的绝对值，入参可为 <code>int</code> , <code>float</code> , <code>complex</code> 型，为一个内置函数
<code>fabs(x)</code>	返回一个数字的绝对值，入参仅可为 <code>int</code> 或 <code>float</code> 型，位于 <code>math</code> 模组中
<code>ceil(x)</code>	返回数字的上入整数，位于 <code>math</code> 模组中
<code>floor(x)</code>	返回数字的下舍整数，位于 <code>math</code> 模组中
<code>exp(x)</code>	返回 <code>e</code> 的 <code>x</code> 次幂 (e^x)，位于 <code>math</code> 模组中
<code>log(x)</code>	返回以 <code>e</code> 为底的指数, i.e. $\ln(x)$ ，位于 <code>math</code> 模组中
<code>log10(x)</code>	返回以 <code>10</code> 为底的指数, i.e. $\log_{10}(x)$ ，位于 <code>math</code> 模组中
<code>modf(x)</code>	返回一个由 <code>x</code> 的小数与整数组成的元组，整数以浮点型表示；位于 <code>math</code> 模组中
<code>max(x1,x2,...)</code>	返回给定参数的最大值，参数可以为序列；为一个内置函数
<code>min(x1,x2,...)</code>	返回给定参数的最小值，参数可以为序列；为一个内置函数
<code>pow(x, y)</code>	返回 <code>x**y</code> 运算 (幂运算) 后的值，为一个内置函数
<code>sqrt(x)</code>	返回数字 <code>x</code> 的平方根，返回值是 <code>float</code> 型，位于 <code>math</code> 模组中
<code>round(x [,n])</code>	返回浮点数 <code>x</code> 的四舍五入值；若给定 <code>n</code> ，则舍入到小数点后 <code>n</code> 位；内置函数
<code>cmp(x, y)</code>	已弃用, 可用 <code>(x>y)-(x<y)</code> 替换。若 <code>x<y</code> ，返回 -1； <code>x==y</code> ，返回 0； <code>x>y</code> ，返回 1.

4.4.4. 随机数函数

Function	Description
<code>choice(seq)</code>	从序列的元素中随机挑选一个元素
<code>randrange([start,] stop [, step])</code>	从指定范围内递增的集合中获取一个随机数
<code>random()</code>	随机生成下一个在 [0,1) 范围内的实数
<code>seed([x])</code>	改变随机数生成器的种子seed
<code>shuffle(list)</code>	将列表的所有元素随机排序
<code>uniform(x, y)</code>	随机生成下一个 [x, y] 范围内的实数
<code>randint(x, y)</code>	随机生成下一个 [x, y] 范围内的整数
<code>sample(seq, length)</code>	从指定的序列中随即的截取指定长度的片段，不修改原序列

注意：以上函数的使用都需先导入 `random` 模块！

4.4.5. 三角函数

Function	Description
<code>acos(x)</code>	返回 x 的反余弦弧度值
<code>asin(x)</code>	返回 x 的正弦弧度值
<code>atan(x)</code>	返回 x 的正切弧度值
<code>atan2(y, x)</code>	返回给定的 x 及 y 坐标值的反正切弧度值
<code>cos(x)</code>	返回 x 弧度的余弦值
<code>sin(x)</code>	返回 x 弧度的正弦值
<code>tan(x)</code>	返回 x 弧度的正切值
<code>hypot(x, y)</code>	返回欧几里德范数： <code>sqrt(x*x+y*y)</code>
<code>degrees(x)</code>	将弧度转换为角度
<code>radians(x)</code>	将角度转换为弧度

注意：以上函数的使用都需先导入 `math` 模块！

4.4.6. 数学常量

Constant	Description
<code>pi</code>	数学常量 π ，圆周率，需导入 <code>math</code> 模块
<code>e</code>	数学常量 e ，自然常数，需导入 <code>math</code> 模块

4.4.7. 其他说明

1. 使用 `round()` 函数时遵循“四舍六入五看齐，奇进偶不进”的规则。
2. Python 3 中舍弃了 `cmp()` 函数，可用 `operator` 模块中的函数替代：

```
1 import operator          #首先要导入运算符模块
2 operator.gt(1,2)         #意思是greater than (大于)
3 operator.ge(1,2)         #意思是greater and equal (大于等于)
4 operator.eq(1,2)         #意思是equal (等于)
5 operator.le(1,2)         #意思是less and equal (小于等于)
6 operator.lt(1,2)         #意思是less than (小于)
```

3. Python 中的 `Fraction` 模块提供了分数类型的支持。

可以同时提供分子(numerator)和分母(denominator)给构造函数用于实例化 `Fraction` 类，但两者必须同时是 `int` 或 `numbers.Rational` 类型，否则抛出类型错误。当分母为 0 初始化时会抛出异常 `ZeroDivisionError`。

```
1 >>> from fractions import Fraction
2 >>> x = Fraction(1,3)
3 >>> y = Fraction(4,6)
4 >>> x + y
5 Fraction(1, 1)
6 >>> Fraction('.25')
7 Fraction(1, 4)
```

浮点数与分数的转换：

```
1 >>> f = 2.5
2 >>> z = Fraction(*f.as_integer_ratio())
3 >>> z
4 Fraction(5, 2)
5 >>> x = Fraction(1,3)
6 >>> float(x)
7 0.3333333333333333
```

4. Python 中的 `decimal` 模块提供了一个 `Decimal` 数据类型用于浮点数计算，拥有更高的精度：

```
1 >>> import decimal
2 >>> decimal.getcontext().prec = 4          # 指定精度 (4位小数)
3 >>> decimal.Decimal(1) / decimal.Decimal(7)
4 Decimal('0.1429')
5 >>> with decimal.localcontext() as ctx :    # 小数上下文管理器
6 ...     ctx.prec = 2
7 ...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
8 ...
9 Decimal('0.33')
10 >>> from decimal import Decimal
11 >>> Decimal.from_float(1.05)
12 Decimal('1.0500000000000000444089209850062616169452667236328125')
```

4.5. 字符串 (String)

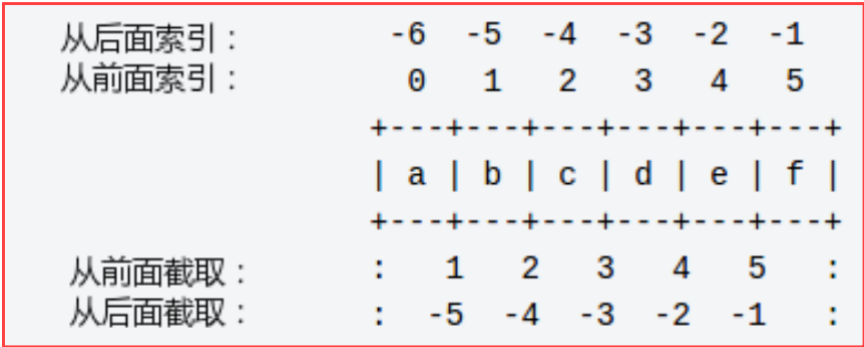
Python中字符串不可以发生改变。
Python中没有单独的字符类型，一个字符就是长度为1的字符串。
Python中单引号和双引号使用完全相同。
Python 中可使用三引号 (''' 或 """) 可以指定一个多行字符串。

4.5.1. 字符串的截取

字符串的截取语法格式如下：

```
1 | 字符串变量[头下标:尾下标:步长]
```

索引值以 0 为开始值，-1 为从末尾的开始位置。



4.5.2. 字符串的操作

Python 中字符串运算符表：

Operator	Description	Example
+	字符串连接	a + b
*	重复输出字符串	a*2
[]	通过 index 索引获取字符串中的字符	a[1]
[:]	截取字符串中的一部分，遵循左闭右开原则	a[1:4]
in	成员运算符，若字符串中包含给定的字符，则返回 True	'H' in a
not in	成员运算符，若字符串中不包含给定的字符，则返回 True	'H' not in a
r/R	原始字符串：字符串直接使用，不转义特殊或不能打印的字符	a = r'\n'
%	格式字符串	

实例：

```
1 | #!/usr/bin/env python3
2 | str = 'Runoob'
3 | print (str) # 输出字符串
4 | print (str[0:-1]) # 输出第一个到倒数第二个的所有字符
5 | print (str[0]) # 输出字符串第一个字符
6 | print (str[2:5]) # 输出从第三个开始到第五个的字符
7 | print (str[2:]) # 输出从第三个开始的后的所有字符
8 | print (str * 2) # 输出字符串两次
9 | print (str + "TEST") # 连接字符串
```

执行以上程序会输出如下结果：

```
1 Runoob
2 Runoo
3 R
4 noo
5 noob
6 RunoobRunoob
7 RunoobTEST
```

4.5.3. 字符串的格式化

4.5.3.1. 使用"%"操作符

操作符" %"的使用格式为：

```
1 %[(name)][flags][width][.precision]typecode
```

- **(name)** 为命名，可不指定；
- **flags** 可以有 `+`, `-`, `空格` 或 `0`，分别表示右对齐, 左对齐, 在正数左侧填充一个空格, 及使用 `0` 填充；
- **width** 表示显示的最小总宽度；
- **precision** 表示小数点后的精度；
- **typecode** 必须指定，不可缺少。

Python 中字符串格式化 **typecode** 表：

Symbol	Description	Symbol	Description
<code>%c</code>	格式化单个字符及其ASCII码	<code>%s</code>	格式化字符串 (采用 <code>str()</code> 的显示)
<code>%d</code> 或 <code>%i</code>	格式化十进制整数	<code>%r</code>	格式化字符串 (采用 <code>repr()</code> 的显示)
<code>%u</code>	格式化无符号整型	<code>%x</code>	格式化无符号十六进制数
<code>%o</code>	格式化无符号八进制数	<code>%X</code>	格式化无符号十六进制数（大写）
<code>%e</code>	用科学计数法格式化浮点数	<code>%f</code>	格式化浮点数字，可指定小数点后的精度
<code>%E</code>	作用同 <code>%e</code> ，用科学计数法格式化浮点数	<code>%g</code>	<code>%f</code> 和 <code>%e</code> 的简写
<code>%%</code>	格式化百分号 <code>%</code> 输出	<code>%G</code>	<code>%f</code> 和 <code>%E</code> 的简写

Python 中格式化操作符**辅助指令**：

Symbol	Function
-	用做左对齐
+	在正数前面显示加号(+)
<space>	在正数前面显示空格
0	显示的数字前面填充'0'而不是默认的空格
m.n	m 是显示的最小总宽度，n 是小数点后的位数
*	定义宽度或者小数点精度 (可实现动态带入)
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X' (取决于用的是'x'还是'X')
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)

实例：

```
1 >>> print("%6.3f" % 2.3)
2 2.300
```

- 第一个 % 后面的内容为显示的格式说明，6 为显示宽度，3 为小数点位数，f 为浮点数类型
- 第二个 % 后面为显示的内容来源，输出结果右对齐，2.300 长度为 5，故前面有一空格

上面的width, precision为两个整数。我们可以利用" *"来动态代入这两个量。如：

```
1 >>> print("%10.*f" % (4, 1.2))
2 1.2000
```

4.5.3.2. 使用str.format()

Python 2.6 开始，新增了一种格式化字符串的函数 `str.format()`，它增强了字符串格式化的功能。基本语法是通过 `{}` 和 `:` 来代替以前的 `%`。`format()` 函数可以接受不限个数的参数，位置可以不按顺序。

```
1 >>> "{} {}".format("hello", "world")      # 不设置指定位置，按默认顺序
2 'hello world'
3 >>> "{0} {1}".format("hello", "world")    # 设置指定位置
4 'hello world'
5 >>> "{1} {0} {1}".format("hello", "world") # 设置指定位置
6 'world hello world'
```

也可设置参数：

```

1  #!/usr/bin/env python3
2  print("网站名 : {name}, 地址 {url}".format(name="菜鸟教程", url="www.runoob.com"))
3
4  # 通过字典设置参数
5  site = {"name": "菜鸟教程", "url": "www.runoob.com"}
6  print("网站名 : {name}, 地址 {url}".format(**site))
7
8  # 通过列表索引设置参数
9  my_list = ['菜鸟教程', 'www.runoob.com']
10 print("网站名 : {0[0]}, 地址 {0[1]}".format(my_list)) # "0" 是必须的

```

也可以向 `str.format()` 传入对象：

```

1  #!/usr/bin/env python3
2  class AssignValue(object) :
3      def __init__(self, value) :
4          self.value = value
5
6  my_value = AssignValue(6)
7  print('value 为: {0.value}'.format(my_value)) # "0" 是可选的
8  print('value 为: {.value}'.format(my_value)) # "0" 是可选的 (两结果相同)

```

如果有个实在很长的格式化字符串，又不想分割它。假设可以用命名来引用被格式化的变量而不是位置，则可以传入一个字典，并用中括号 (`[]`) 访问它的键：

```

1  >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
2  >>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
3        'Dcab: {0[Dcab]:d}'.format(table))
4  Jack: 4098; Sjoerd: 4127; Dcab: 8637678

```

也可以用 `**` 标志将这个字典以关键字参数的方式传入：

```

1  >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
2  >>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
3  Jack: 4098; Sjoerd: 4127; Dcab: 8637678

```

数字的格式化 (大括号中使用 `:` 来表示数字的格式化，`:` 前仍可添加位置参数，如 `"0:"`)：

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	保留小数点后两位
3.1415926	{:+.2f}	+3.14	带符号保留小数点后两位
-1	{:+.2f}	-1.00	带符号保留小数点后两位
2.71828	{:.0f}	3	不带小数
5	{:0>2d}	05	数字补零 (填充左边, 宽度为2)
5	{:x<4d}	5xxx	数字补x (填充右边, 宽度为4)
1000000	{:,}	1,000,000	以逗号分隔的数字格式
0.25	{:.2%}	25.00%	百分比格式
1000000000	{:.2e}	1.00e+09	指数记法
13	{:>10d}	13	右对齐 (默认, 宽度为10)
13	{:<10d}	13	左对齐 (宽度为10)
13	{:^10d}	13	中间对齐 (宽度为10)
11	<code>'{:b}'.format(11)</code>	1011	二进制
11	<code>'{:d}'.format(11)</code>	11	十进制
11	<code>'{:o}'.format(11)</code>	13	八进制
11	<code>'{:x}'.format(11)</code>	b	十六进制
11	<code>'{:#x}'.format(11)</code>	0xb	十六进制前面显示'0x'
11	<code>'{:#X}'.format(11)</code>	0XB	十六进制前面显示'0X'

总结：{} 中的格式样式为：

```

1  {[name][:][symbol][align][width][.precision][type]}
2
3  name      : 位置参数 (可为数字, 键值, 索引, .value 等)
4  symbol    : 填充符号 (只能是一个字符, 如: 正号, 逗号, 0, x 等; 不指定则默认使用空格填充)
5  align     : 居中(^)、左对齐(<)、右对齐(>)
6  width     : 表示总宽度
7  .precision: 表示小数点后的位数
8  typecode  : f, d, e, E, g, G, %, b, d, o, #o, x, #x, #X

```

1. "^, <, >" 分别是**居中**、**左对齐**、**右对齐**, 左边紧跟的符号代表补齐方式, 右边紧跟的数字代表宽度;
2. ":" 号后面带填充的字符, 只能是一个字符, 不指定则默认是用空格填充;
3. "+" 表示在正数前显示正号负数前显示负号; (空格) 表示在正数前加空格; ",", 表示用逗号分隔;
4. 此外可以使用大括号 {} 来转义大括号, 如下实例:

```

1  >>> print("{} 对应的位置是 {}".format("runoob"))
2  runoob 对应的位置是 {}

```

4.5.3.3. 使用 f-string 来格式化

Python 3.6 之后添加了 **f-string**, 称之为**字面量格式化字符串**, 是新的格式化字符串的语法。其字符串以 **f** 开头, 后面跟着字符串, 字符串中的表达式用大括号 {} 包起来, 它会将**变量**或**表达式**计算后的值替换进去。实例:

```

1  >>> name = 'Runoob'
2  >>> f'Hello {name}' # 替换变量
3  'Hello Runoob'
4  >>> f'{1+2}'      # 使用表达式
5  '3'
6  >>> w = {'name': 'Runoob', 'url': 'www.runoob.com'}
7  >>> f'{w["name"]}: {w["url"]}'
8  'Runoob: www.runoob.com'

```

此外, 在 Python 3.8 版本中可以使用 = 符号来拼接运算表达式与结果:

```

1  >>> x = 1
2  >>> print(f'{x+1}') # Python 3.6
3  2
4  >>> x = 1
5  >>> print(f'{x+1=}') # Python 3.8
6  'x+1=2'

```

4.5.4. 字符串的转义

Python 使用反斜杠 (\) 来转义; 使用 **r** 或 **R** 可以让反斜杠不发生转义, 表示原始字符串:

```

1  >>> print('Ru\noob')
2  Ru
3  oob
4  >>> print(r'Ru\noob')
5  Ru\noob

```

Python 中转义字符表：

Escape Character	Description	Escape Character	Description
<code>\</code> (at the end line)	续行符	<code>\\</code>	反斜杠符号
<code>\'</code>	单引号	<code>\''</code>	双引号
<code>\a</code>	响铃	<code>\b</code>	退格
<code>\000</code>	空	<code>\n</code>	换行
<code>\v</code>	纵向制表符	<code>\t</code>	横向制表符
<code>\r</code>	回车	<code>\f</code>	换页
<code>\oyy</code>	八进制数, <code>yy</code> 代表字符	<code>\xyy</code>	十六进制数, <code>yy</code> 代表字符
<code>\other</code>	其他的字符以普通格式输出		

4.5.5. 字符串的内建函数

Python 的字符串常用内建函数如下：

No.	Function & Description
1	<code>capitalize()</code> 将字符串的首字符转换为大写, 其余字符转换为小写; 若首字符非字母, 则首字符不转换
2	<code>center(width, fillchar)</code> 返回一个指定的宽度 <code>width</code> 居中的字符串, <code>fillchar</code> 为填充的字符 (默认为空格)
3	<code>count(str, beg=0, end=len(string))</code> 返回 <code>str</code> 在 <code>string</code> 里面出现的次数, 若 <code>beg</code> 或 <code>end</code> 指定则返回指定范围内 <code>str</code> 出现的次数
4	<code>bytes.decode(encoding="utf-8", errors="strict")</code> Python 3 中没有 <code>decode</code> 方法, 但我们可以使用 <code>bytes</code> 对象的 <code>decode()</code> 方法来解码给定的 <code>bytes</code> 对象, 这个 <code>bytes</code> 对象可以由 <code>str.encode()</code> 来编码返回
5	<code>encode(encoding='UTF-8', errors='strict')</code> 以 <code>encoding</code> 指定的编码格式编码字符串, 如果出错默认报一个 <code>ValueError</code> 的异常, 除非 <code>errors</code> 指定的是 <code>ignore</code> 或 <code>replace</code>
6	<code>endswith(suffix, beg=0, end=len(string))</code> 检查字符串是否以 <code>obj</code> 结束, 如果 <code>beg</code> 或者 <code>end</code> 指定则检查指定的范围内是否以 <code>obj</code> 结束, 如果是返回 <code>True</code> , 否则返回 <code>False</code> .
7	<code>expandtabs(tabsize=8)</code> 把字符串 <code>string</code> 中的 <code>tab</code> 符号转为空格, <code>tab</code> 符号默认的空格数是 8。
8	<code>find(str, beg=0, end=len(string))</code> 检测 <code>str</code> 是否包含在字符串中, 如果指定范围 <code>beg</code> 和 <code>end</code> , 则检查是否包含在指定范围内, 如果包含返回开始的索引值, 否则返回 -1
9	<code>index(str, beg=0, end=len(string))</code> 跟 <code>find()</code> 方法一样, 只不过如果 <code>str</code> 不在字符串中会报一个异常.
10	<code>isalnum()</code> 如果字符串至少有一个字符并且所有字符都是字母或数字则返回 <code>True</code> , 否则返回 <code>False</code>
11	<code>isalpha()</code> 如果字符串至少有一个字符并且所有字符都是字母则返回 <code>True</code> , 否则返回 <code>False</code>
12	<code>isdigit()</code> 如果字符串只包含数字则返回 <code>True</code> 否则返回 <code>False</code> ..
13	<code>islower()</code> 如果字符串中包含至少一个区分大小写的字符, 并且所有这些(区分大小写的)字符都是小写, 则返回 <code>True</code> , 否则返回 <code>False</code>
14	<code>isnumeric()</code> 如果字符串中只包含数字字符, 则返回 <code>True</code> , 否则返回 <code>False</code>
15	<code>isspace()</code> 如果字符串中只包含空白, 则返回 <code>True</code> , 否则返回 <code>False</code> .
16	<code>istitle()</code> 如果字符串是标题化的(见 <code>title()</code>)则返回 <code>True</code> , 否则返回 <code>False</code>
17	<code>isupper()</code> 如果字符串中包含至少一个区分大小写的字符, 并且所有这些(区分大小写的)字符都是大写, 则返回 <code>True</code> , 否则返回 <code>False</code>
18	<code>join(seq)</code> 以指定字符串作为分隔符, 将 <code>seq</code> 中所有元素(的字符串表示)合并为一个新的字符串
19	<code>len(string)</code> 返回字符串长度
20	<code>ljust(width[, fillchar])</code> 返回一个原字符串左对齐, 并使用 <code>fillchar</code> 填充至长度 <code>width</code> 的新字符串, <code>fillchar</code> 默认为空格。
21	<code>lower()</code> 转换字符串中所有大写字符为小写.

No.	Function & Description
22	lstrip() 截掉字符串左边的空格或指定字符。
23	maketrans() 创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
24	max(str) 返回字符串 str 中最大的字母。
25	min(str) 返回字符串 str 中最小的字母。
26	replace(old, new[, max]) 把 将字符串中的 str1 替换成 str2,如果 max 指定，则替换不超过 max 次。
27	rfind(str, beg=0, end=len(string)) 类似于 find()函数，不过是从右边开始查找。
28	rindex(str, beg=0, end=len(string)) 类似于 index()，不过是从右边开始。
29	rjust(width[, fillchar]) 返回一个原字符串右对齐,并使用fillchar(默认空格) 填充至长度 width 的新字符串
30	rstrip() 删除字符串字符串末尾的空格。
31	split(str="", num=string.count(str)) num=string.count(str)) 以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num+1 个子字符串
32	splitlines([keepends]) 按照行('\r', '\r\n', '\n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
33	startswith(substr, beg=0, end=len(string)) 检查字符串是否是以指定子字符串 substr 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查。
34	strip([chars]) 在字符串上执行 lstrip()和 rstrip()
35	swapcase() 将字符串中大写转换为小写，小写转换为大写
36	title() 返回"标题化"的字符串,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
37	translate(table, deletechars="") 根据 str 给出的表(包含 256 个字符)转换 string 的字符,要过滤掉的字符放到 deletechars 参数中
38	upper() 转换字符串中的小写字母为大写
39	zfill(width) 返回长度为 width 的字符串，原字符串右对齐，前面填充0
40	isdecimal() 检查字符串是否只包含十进制字符，如果是返回 true，否则返回 false。

补充说明：

1. **str.isdigit**、**str.decimal** 和 **str.isnumeric** 区别

Comparison

`str.isdigit()`

True: Unicode数字, byte数字(单字节), 全角数字(双字节)

False: 汉字数字, 罗马数字, 小数

Error: 无

`str.isdecimal()`

True: Unicode数字, 全角数字(双字节)

False: 汉字数字, 罗马数字, 小数

Error: byte数字(单字节)

`str.isdigit()`

True: Unicode数字, 全角数字(双字节), 汉字数字

False: 罗马数字, 小数

Error: byte数字(单字节)

```
1  num = "1" # unicode
2  num.isdigit() # True
3  num.isdecimal() # True
4  num.isnumeric() # True
5
6  num = "一" # 全角
7  num.isdigit() # True
8  num.isdecimal() # True
9  num.isnumeric() # True
10
11 num = b"1" # byte
12 num.isdigit() # True
13 num.isdecimal() # AttributeError 'bytes' object has no attribute 'isdecimal'
14 num.isnumeric() # AttributeError 'bytes' object has no attribute 'isnumeric'
15
16 num = "IV" # 罗马数字
17 num.isdigit() # False
18 num.isdecimal() # False
19 num.isnumeric() # False
20
21 num = "四" # 汉字
22 num.isdigit() # False
23 num.isdecimal() # False
24 num.isnumeric() # True
```

2. 针对 `Counter` 的升级使用，示例如下：

```
1  #必须引用如下库
2  from collections import Counter
3
4  #定义两个字符串变量
5  Var1 = "1116122137143151617181920849510"
6  Var2 = "1987262819009787718192084951"
7
8  #以字典的形式，输出每个字符串中出现的字符及其数量
9  print (Counter(Var1))
10 print (Counter(Var2))
```

输出如下：

```
1 Counter({'1': 12, '2': 3, '6': 2, '3': 2, '7': 2, '4': 2, '5': 2, '8': 2, '9':  
2, '0': 2})  
2 Counter({'1': 5, '9': 5, '8': 5, '7': 4, '2': 3, '0': 3, '6': 1, '4': 1, '5':  
1})
```

3. 字符串与列表，元组的互相转换：

```
1 # 1、字符串转换为列表  
2 var='菜鸟教程'  
3 list=[]  
4 list=[i for i in var]  
5  
6 # 2、列表转化为字符串  
7 var1=''.join(list)  
8  
9 # 3、字符串转化为元组，使用 tuple() 函数  
10 tup=tuple(var)
```

4.6. 列表 (List)

List (列表) 是 Python 中使用最频繁的数据类型，可以完成大多数集合类的数据结构实现。

列表是写在方括号 [] 之间、用逗号分隔开的元素列表。其元素的类型可以不相同，支持数字, 字符串, 元组等，甚至可以包含列表 (即列表的嵌套)。

```
1 # 创建长度为10的空列表  
2 list_empty = [None]*10  
3  
4 # 创建二维列表，将需要的参数写入 cols 和 rows 即可  
5 list_2d = [[0 for col in range(cols)] for row in range(rows)]
```

列表推导式书写形式：

```
1 [表达式 for 变量 in 列表]  
2 或者  
3 [表达式 for 变量 in 列表 if 条件]
```

实例：

```
1 #!/usr/bin/env python3  
2 li = [1,2,3,4,5,6,7,8,9]  
3 print ([x**2 for x in li])  
4 print ([x**2 for x in li if x>5])  
5 print (dict([(x,x*10) for x in li]))  
6 print ([ (x, y) for x in range(10) if x % 2 if x > 3 for y in range(10) if y > 7 if  
7 y != 8])  
8  
9 vec =[2,4,6]  
10 vec2=[4,3,-9]  
11 sq = [vec[i]+vec2[i] for i in range(len(vec))]  
12 print (sq)  
13 print ([x*y for x in [1,2,3] for y in [1,2,3]])
```

```

14 testList = [1,2,3,4]
15 def mul2(x):
16     return x*2
17
18 print ([mul2(i) for i in testList])

```

输出结果：

```

1 [1, 4, 9, 16, 25, 36, 49, 64, 81]
2 [36, 49, 64, 81]
3 {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60, 7: 70, 8: 80, 9: 90}
4 [(5, 9), (7, 9), (9, 9)]
5 [6, 7, -3]
6 [1, 2, 3, 2, 4, 6, 3, 6, 9]
7 [2, 4, 6, 8]

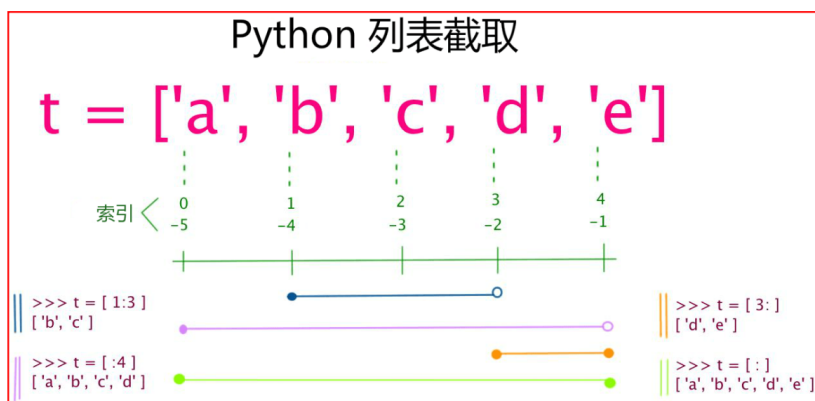
```

4.6.1. 列表的截取

列表截取的语法格式如下：

```
1 变量[头下标:尾下标:步长]
```

索引值以 0 为开始值，-1 为从末尾的开始位置。



Python 列表截取可以接收第三个参数，作用是截取的步长。

```

0      1      2      3      4      5      6
>>> letters = ['c', 'h', 'e', 'c', 'k', 'i', 'o']

                2
>>> letters[1:4:2]
['h', 'c']

```

若步长参数为负数则表示逆向读取，以下实例用于翻转字符串：

```

1  #!/usr/bin/env python3
2  def reverseWords(input):
3      # 通过空格将字符串分隔符，把各个单词分隔为列表
4      inputWords = input.split(" ")
5      # 翻转字符串
6      # 假设列表 list = [1,2,3,4],
7      # list[0]=1, list[1]=2, 而-1表示最后一个元素 list[-1]=4 (与list[3]一样)

```

```

8      # inputWords[-1::-1] 有三个参数
9      # 第一个参数 -1 表示最后一个元素
10     # 第二个参数为空，表示移动到列表末尾
11     # 第三个参数为步长，-1 表示逆向
12     inputWords=inputWords[-1::-1]
13     # 重新组合字符串
14     output = ' '.join(inputWords)
15     return output
16
17 if __name__ == "__main__":
18     input = 'I like runoob'
19     rw = reverseWords(input)
20     print(rw)

```

4.6.2. 列表的操作

Python Expression	Output	Description
<code>len([1, 2, 3])</code>	3	长度
<code>[1, 2, 3] + [4, 5, 6]</code>	[1, 2, 3, 4, 5, 6]	组合
<code>['Hi!'] * 7</code>	['Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
<code>3 in [1, 2, 3]</code>	True	元素是否存在于列表中
<code>for x in [1, 2, 3]: print(x, end=" ")</code>	1 2 3	迭代

加号 + 是列表连接符；星号 * 表示重复操作，紧跟的数字为重复的次数。

```

1  #!/usr/bin/env python3
2  list = [ 'abcd', 786 , 2.23, 'runoob', 70.2 ]
3  tinylist = [123, 'runoob']
4  print (list) # 输出完整列表
5  print (list[0]) # 输出列表第一个元素
6  print (list[1:3]) # 从第二个开始输出到第三个元素
7  print (list[2:]) # 输出从第三个元素开始的所有元素
8  print (tinylist * 2) # 输出两次列表
9  print (list + tinylist) # 连接列表

```

以上程序的输出结果为：

```

1  ['abcd', 786, 2.23, 'runoob', 70.2]
2  abcd
3  [786, 2.23]
4  [2.23, 'runoob', 70.2]
5  [123, 'runoob', 123, 'runoob']
6  ['abcd', 786, 2.23, 'runoob', 70.2, 123, 'runoob']

```

与 Python 中字符串类型不一样的是，列表中的元素是可以改变的：

```

1  >>>a = [1, 2, 3, 4, 5, 6]
2  >>> a[0] = 9
3  >>> a[2:5] = [13, 14, 15]
4  >>> a
5  [9, 2, 13, 14, 15, 6]
6  >>> a[2:5] = [] # 将元素的值设为空[]即可删除相应元素
7  >>> a
8  [9, 2, 6]

```

注意：

```

1  >>> list = [ 'abcd', 786 , 2.23, 'runoob', 70.2 ]
2  >>> print (list[2])
3  2.23
4  >>> print (list[2:3])
5  [2.23]

```

这两句话打印的内容其实是相似的，但输出的结果属于不同的类型：

```

1  >>> a = list[2]
2  >>> b = list[2:3]
3  >>> type(a)
4  <class 'float'>
5  >>> type(b)
6  <class 'list'>

```

4.6.3. 列表的函数与方法

Python 包含以下内置函数：

No.	Function	No.	Function
1	len(list) 计算列表元素个数	3	min(list) 返回列表元素的最小值
2	max(list) 返回列表元素的最大值	4	list(seq) 将元组转换为列表

Python 包含以下方法：

序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
4	list.index(obj) 从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj) 将对象插入列表
6	list.pop([index=-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
7	list.remove(obj) 移除列表中某个值的第一个匹配项
8	list.reverse() 反向列表中元素
9	list.sort(key=None, reverse=False) 对原列表进行排序
10	list.clear() 清空列表
11	list2 = list1.copy() 复制列表

注1：列表发生改变后的说明：

1. 通过 `list.clear()`，`list.remove()`，`list.pop()`，`list.append()` 等方法改变列表之后，相应的已经赋值给其他变量的列表也会发生改变。

```

1  >>> d = {} # 空字典
2  >>> s = s = ['一', '二', '一', '三', '四']
3  >>> d["大写数字"] = s
4  >>> d
5  {'大写数字': ['一', '二', '一', '三', '四']}
6  >>> s.remove('一')
7  >>> d
8  {'大写数字': ['二', '一', '三', '四']}
9  >>> s.pop()
10 >>> d
11 {'大写数字': ['二', '一', '三']}
12 >>> s.append('五')
13 >>> d
14 {'大写数字': ['二', '一', '三', '五']}
15 >>> s.clear()
16 >>> d
17 {'大写数字': []}

```

注2：列表排序 `list.sort()` 方法的说明：

1. 数值型排序：

```

1  #原始列表
2  a=[[1,3],[3,2],[2,4],[1,2],[1,5],[2,5]]
3  print(a)
4  #先按第一个元素升序排序，第一个元素相同按第二个元素升序排序
5  a.sort()

```



```

6 print(a)
7 #先按第一个元素升序排序，第一个元素相同则保持原来的顺序
8 a=[[1,3],[3,2],[2,4],[1,2],[1,5],[2,5]]
9 a.sort(key=lambda x:x[0])
10 print(a)
11 #先按第二个元素升序排序，第二个元素相同则保持原来的顺序
12 a=[[1,3],[3,2],[2,4],[1,2],[1,5],[2,5]]
13 a.sort(key=lambda x:x[1])
14 print(a)
15 #先按第二个元素升序排序，第二个元素相同按第一个元素升序排序
16 a.sort(key=lambda x:(x[1],x[0]))
17 print(a)
18 #先按第二个元素升序排序，第二个元素相同按第一个元素降序排序
19 a.sort(key=lambda x:(x[1],-x[0]))
20 print(a)

```

2. 字符串排序：

```

1 #原始列表
2 a=['delphi', 'Delphi', 'python', 'Python', 'golang', 'Golang', 'c++', 'C++',
   'c', 'C']
3 print(a)
4 #按字典序升序排序
5 a.sort()
6 print(a)
7 #按字典序降序排序
8 a.sort(reverse=True)
9 print(a)
10 #按字符串长度升序排列
11 a.sort(key=lambda ele:len(ele))
12 print(a)
13 #按字符串长度降序排列
14 a.sort(key=lambda ele:len(ele),reverse=True)
15 print(a)
16 #先按字符串长度升序排序，长度相同按字典序升序排序
17 a.sort(key=lambda x:(len(x),x))
18 print(a)
19 #先按字符串长度升序排序，长度相同按字典序降序排序
20 a.sort(key=lambda x:(len(x),list(map(lambda c:-ord(c),x))))
21 print(a)
22 #先按字符串长度降序排序，长度相同按字典序升序排序
23 a.sort(key=lambda x:(-len(x),x))
24 print(a)
25 #先按字符串长度降序排序，长度相同按字典序降序排序
26 a.sort(key=lambda x:(-len(x),list(map(lambda c:-ord(c),x))))
27 print(a)

```

3. 根据列表中类对象的属性排序:

```

1 #!/usr/bin/env python3
2 class element(object):
3     def __init__(self,id="",name=""):
4         self.id=id
5         self.name=name
6
7     def __lt__(self, other): # override < 操作符

```

```

8         if self.id < other.id:
9             return True
10        return False
11
12        def __str__(self): # override __str__
13            return "id={0},name={1}".format(self.id,self.name)
14
15        def sort_by_attribute():
16            list=[element(id="130",name="json"),
17                  element(id="01",name="jack"),
18                  element(id="120",name="tom")]
19            list.sort()
20            for item in list:
21                print(item)
22
23        if __name__ == "__main__" :
24            sort_by_attribute()

```

由于 `list.sort()` 函数在排序时，使用的是小于号对比，所以自定义的数据类型需要 **override lt (小于)** 函数才能实现排序。以上代码根据 `element` 的 `id` 属性升序排列。

4. 动态的根据用户指定的索引进行排序:

```

1  #!/usr/bin/env python3
2  def two_d_list_sort2(sort_index="0,1,2"): # 动态的根据传入的元素索引进行排序
3      list=[ ["1","c++","demo"],
4              ["1","c","test"],
5              ["2","java","sli"],
6              ["8","golang","google"],
7              ["4","python","gil"],
8              ["5","swift","apple"]
9      ]
10     key_set=""
11     for item in sort_index.split(","):
12         key_set+="ele["+item+"]+"
13     key_set=key_set.rstrip("+")
14     list.sort(key=lambda ele:eval(key_set))
15     print("排序索引:",sort_index,list)
16
17     if __name__=="__main__":
18         two_d_list_sort2("0")
19         two_d_list_sort2("1")
20         two_d_list_sort2("2")
21         two_d_list_sort2("0,1")

```

有时候，在写代码之前并不知道根据二维表的哪几列排序，而是在程序运行的时候根据输入或配置决定的，为了解决这个动态根据多个列或属性排序的问题，可借助 **eval()** 函数把字符串编译成 Python 代码并运行，从而达到动态根据多个列或属性排序的目的。排序结果：

```

1  排序索引: 0 [['1', 'c++', 'demo'], ['1', 'c', 'test'], ['2', 'java', 'sli'],
   ['4', 'python', 'gil'], ['5', 'swift', 'apple'], ['8', 'golang', 'google']]
2  排序索引: 1 [['1', 'c', 'test'], ['1', 'c++', 'demo'], ['8', 'golang',
   'google'], ['2', 'java', 'sli'], ['4', 'python', 'gil'], ['5', 'swift',
   'apple']]
3  排序索引: 2 [['5', 'swift', 'apple'], ['1', 'c++', 'demo'], ['4', 'python',
   'gil'], ['8', 'golang', 'google'], ['2', 'java', 'sli'], ['1', 'c', 'test']]
4  排序索引: 0,1 [['1', 'c', 'test'], ['1', 'c++', 'demo'], ['2', 'java', 'sli'],
   ['4', 'python', 'gil'], ['5', 'swift', 'apple'], ['8', 'golang', 'google']]

```

4.7. 元组 (Tuple)

Tuple (元组) 与列表类似，不同之处在于**元组的元素不能修改**。

元组写在小括号 () 里，元素之间用逗号隔开。元组中的元素类型也可以不相同。

构造**包含 0 个或 1 个元素**的元组比较特殊，所以有一些额外的语法规则：

```

1  tup1 = ()      # 空元组
2  tup2 = (20,)   # 一个元素，需要在元素后添加逗号

```

关于元组是不可变的：所谓元组的不可变指的是元组所指向的内存中的内容不可变。

```

1  >>> tup = ('r', 'u', 'n', 'o', 'o', 'b')
2  >>> tup[0] = 'g' # 元组不支持修改元素
3  Traceback (most recent call last):
4    File "<stdin>", line 1, in <module>
5  TypeError: 'tuple' object does not support item assignment
6  >>> id(tup)      # 查看内存地址
7  4440687904
8  >>> tup = (1,2,3)
9  >>> id(tup)
10 4441088800      # 内存地址不一样了

```

从以上实例可以看出，重新赋值的元组 `tup` 绑定到新的对象，而不是修改了原来的对象。

4.7.1. 元组的截取

元组与字符串类似，可以被索引且下标索引从0开始，-1为从末尾开始的位置。

元组也可以进行截取。其实可以把字符串看作一种特殊的元组。

```

1  #!/usr/bin/env python3
2  tuple = ('abcd', 786 , 2.23, 'runoob', 70.2)
3  tinytuple = (123, 'runoob')
4  print (tuple) # 输出完整元组
5  print (tuple[0]) # 输出元组的第一个元素
6  print (tuple[1:3]) # 输出从第二个元素开始到第三个元素
7  print (tuple[2:]) # 输出从第三个元素开始的所有元素
8  print (tinytuple * 2) # 输出两次元组
9  print (tuple + tinytuple) # 连接元组

```

以上程序输出结果：

```

1 ('abcd', 786, 2.23, 'runoob', 70.2)
2 abcd
3 (786, 2.23)
4 (2.23, 'runoob', 70.2)
5 (123, 'runoob', 123, 'runoob')
6 ('abcd', 786, 2.23, 'runoob', 70.2, 123, 'runoob')

```

4.7.2. 元组的操作

虽然元组的元素不可改变，但它可以包含可变的对象，比如 list 列表。

```

1 >>> t = ('a', 'b', ['A', 'B'])
2 >>> t[2][0] = 'X'
3 >>> t[2][1] = 'Y'
4 >>> t
5 ('a', 'b', ['X', 'Y'])

```

元组中的元素值是不允许修改的，但可以对元组进行连接组合：

```

1 #!/usr/bin/env python3
2 tup1 = (12, 34.56);
3 tup2 = ('abc', 'xyz')
4 # 以下修改元组元素操作是非法的。
5 # tup1[0] = 100
6
7 # 创建一个新的元组
8 tup3 = tup1 + tup2;
9 print (tup3)

```

元组中的元素值是不允许删除的，但我们可以使用 `del` 语句来删除整个元组：

```

1 #!/usr/bin/env python3
2 tup = ('Google', 'Runoob', 1997, 2000)
3 print (tup)
4 del tup;

```

元组运算符表：

Python Expression	Output	Description
<code>len((1, 2, 3))</code>	3	计算元素个数
<code>(1, 2, 3) + (4, 5, 6)</code>	(1, 2, 3, 4, 5, 6)	连接组合
<code>('Hi!') * 8</code>	('Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
<code>3 in (1, 2, 3)</code>	True	元素是否存在
<code>for x in (1, 2, 3): print(x, end="")</code>	1 2 3	迭代

4.7.3. 元组的内置函数

Python 中元组包含了以下内置函数：

No.	Function	No.	Function
1	len(tuple) 计算元组元素个数	3	min(tuple) 返回元组中元素最小值
2	max(tuple) 返回元组中元素最大值	4	tuple(seq) 将序列转换为元组

4.8. 集合 (Set)

集合 (set) 是由一个或数个形态各异的大小整体组成的，构成集合的事物或对象称作元素或是成员。集合的基本功能是进行成员关系测试和删除重复元素。可以使用大括号 {} 或者 `set()` 函数创建集合。

注意：创建一个空集合必须用 `set()` 而不是大括号 {}，因为 {} 是用来创建一个空字典的。

创建格式：

```
1 param = {value01, value02, ...}
2 或者
3 set(value01, value02, ...)
```

实例：

```
1 #!/usr/bin/env python3
2 student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}
3 print(student) # 输出集合，重复的元素被自动去掉
4
5 # 成员测试
6 if 'Rose' in student :
7     print('Rose 在集合中')
8 else :
9     print('Rose 不在集合中')
10
11 # set可以进行集合运算
12 a = set('abracadabra')
13 b = set('alacazam')
14 print(a)
15 print(a - b) # a 和 b 的差集
16 print(a | b) # a 和 b 的并集
17 print(a & b) # a 和 b 的交集
18 print(a ^ b) # a 和 b 中不同时存在的元素
```

Python 中集合的内置方法有：

Method	Description
set.add(elmnt)	为集合添加元素 (若已存在则舍弃)
set.update(x)	为集合添加元素，入参 <code>x</code> 可为列表, 元组, 字典等
set.discard(elmnt)	删除集合中指定的元素 (若元素不存在不报错)
set.remove(elmnt)	删除集合中指定的元素 (若元素不存在则报错)
set.pop()	随机地移除一个元素
set.copy()	拷贝一个集合
set.clear()	移除集合中的所有元素
set = set1.difference(set2)	返回一集合中不存在于另一指定集合的元素
set1.difference_update(set2)	移除一集合中存在于一指定集合的元素
set = set1.intersection(set2)	返回一集合中存在于一指定集合的元素
set1.intersection_update(set2)	移除一集合中不存在于一指定集合的元素
set = set1.symmetric_difference(set2)	返回两个集合中不重复的元素组成的新集合
set1.symmetric_difference_update(set2)	返回两个集合中不重复的元素组成的新集合
set = set1.union(set2)	返回两个集合的并集
set1.isdisjoint(set2)	判断两个集合是否包含相同元素，若无返回 <code>True</code> ，否则 <code>False</code>
set1.issubset(set2)	判断指定集合是否该方法参数集合的子集。
set1.issuperset(set2)	判断该方法的参数集合是否为指定集合的子集

实例：

```

1  >>> thisset = set(("Google", "Runoob", "Taobao"))
2  >>> thisset.add("Facebook")
3  >>> print(thisset)
4  {'Taobao', 'Facebook', 'Google', 'Runoob'}
5
6  >>> thisset = set(("Google", "Runoob", "Taobao"))
7  >>> thisset.update({1,3})
8  >>> print(thisset)
9  {1, 3, 'Google', 'Taobao', 'Runoob'}
```

```

10 >>> thisset.update([1,4],[5,6])
11 >>> print(thisset)
12 {1, 3, 4, 5, 6, 'Google', 'Taobao', 'Runoob'}
13
14 >>> thisset = set(("Google", "Runoob", "Taobao"))
15 >>> thisset.remove("Taobao")
16 >>> print(thisset)
17 {'Google', 'Runoob'}
18 >>> thisset.remove("Facebook") # 不存在会发生错误
19 Traceback (most recent call last):
20   File "<stdin>", line 1, in <module>
21   KeyError: 'Facebook'
22 >>>

```

注意：

1. `set.update("字符串")` 与 `set.update({"字符串"})` 或 `set.update(("字符串"))` 含义不同：
 - o `set.update({"字符串"})` 将字符串添加到集合中，有重复的会忽略。
 - o `set.update("字符串")` 将字符串拆分为单个字符后，再一个个添加到集合中，有重复的会忽略。

```

1 >>> thisset = set(("Google", "Runoob", "Taobao"))
2 >>> print(thisset)
3 {'Google', 'Runoob', 'Taobao'}
4 >>> thisset.update({"Facebook"})
5 >>> print(thisset)
6 {'Google', 'Runoob', 'Taobao', 'Facebook'}
7 >>> thisset.update("Yahoo")
8 >>> print(thisset)
9 {'h', 'o', 'Facebook', 'Google', 'Y', 'Runoob', 'Taobao', 'a'}

```

2. 集合用 `set.pop()` 方法删除元素的不一样的感想如下：

对于 list, tuple 类型中的元素，转换集合时会去掉重复的元素，且具有升序排列的功能：

```

1 >>> list = [1,1,2,3,4,5,3,1,4,6,5]
2 >>> set(list)
3 {1, 2, 3, 4, 5, 6}
4 >>> tuple = (2,3,5,6,3,5,2,5)
5 >>> set(tuple)
6 {2, 3, 5, 6}

```

有人认为 `set.pop()` 是随机删除集合中的一个元素，其实不然！对于是字典和字符串转换的集合，确实是随机删除元素的。但当集合是由列表和元组构成时，`set.pop()` 是从左边删除元素的。如下：

```

1 >>> set1 = set([9,4,5,2,6,7,1,8])
2 >>> print(set1)
3 {1, 2, 4, 5, 6, 7, 8, 9}
4 >>> print(set1.pop())
5 1
6 >>> print(set1)
7 {2, 4, 5, 6, 7, 8, 9}

```

4.9. 字典 (Dictionary)

字典 (Dictionary) 是Python中另一个非常有用的内置数据类型。

列表是有序的对象集合，字典是无序的对象集合。

两者之间的区别在于：字典当中的元素是通过键 (key) 来存取的，而不是通过偏移存取。

字典是一种映射类型，用 {} 标识，它是一个无序的 键(key):值(value) 的集合。

注意: 键 (key) 必须使用不可变类型。在同一个字典中，键 (key) 必须是唯一的。

```
1  #!/usr/bin/env python3
2  dict = {} # 创建空字典
3  dict['one'] = "1 - 菜鸟教程"
4  dict[2] = "2 - 菜鸟工具"
5  tinydict = {'name': 'runoob', 'code': 1, 'site': 'www.runoob.com'}
6  print (dict['one']) # 输出键为 'one' 的值
7  print (dict[2]) # 输出键为 2 的值
8  print (tinydict) # 输出完整的字典
9  print (tinydict.keys()) # 输出所有键
10 print (tinydict.values()) # 输出所有值
```

以上实例输出结果：

```
1  1 - 菜鸟教程
2  2 - 菜鸟工具
3  {'name': 'runoob', 'code': 1, 'site': 'www.runoob.com'}
4  dict_keys(['name', 'code', 'site'])
5  dict_values(['runoob', 1, 'www.runoob.com'])
```

构造函数 dict() 可以直接从键值对序列中构建字典，例如：

```
1  >>>dict([('Runoob', 1), ('Google', 2), ('Taobao', 3)])
2  {'Taobao': 3, 'Runoob': 1, 'Google': 2}
3  >>> {x: x**2 for x in (2, 4, 6)}
4  {2: 4, 4: 16, 6: 36}
5  >>> dict(Runoob=1, Google=2, Taobao=3)
6  {'Runoob': 1, 'Google': 2, 'Taobao': 3}
7  >>>
8  >>> dict_1 = dict([('a',1),('b',2),('c',3)]) #元素为元组的列表
9  >>> dict_1
10 {'a': 1, 'b': 2, 'c': 3}
11 >>> dict_2 = dict({'a',1},{'b',2},{'c',3}) #元素为元组的集合
12 >>> dict_2
13 {'b': 2, 'c': 3, 'a': 1}
14 >>> dict_3 = dict(['a',1],['b',2],['c',3]) #元素为列表的列表
15 >>> dict_3
16 {'a': 1, 'b': 2, 'c': 3}
17 >>> dict_4 = dict(('a',1),('b',2),('c',3)) #元素为元组的元组
18 >>> dict_4
19 {'a': 1, 'b': 2, 'c': 3}
```

另外，字典类型也有一些内置的函数，例如 clear()、keys()、values()、items() 等。

```
1  >>> dict1 = {'abc':1,"cde":2,"d":4,"c":567,"d":"key1"}
2  >>> for k,v in dict1.items():
3  ...     print(k,":",v)
```


Python 中的字典是使用了一个称为散列表 (hashtable) 的算法，不管字典中有多少项使用 `in` 操作符花费的时间都差不多。如果把一个字典对象作为 `for` 的迭代对象，那么这个操作将会遍历该字典的键 (key) 而不是其值 (value)：

```
1 def example(d):
2     # d 是一个字典对象
3     for c in d:
4         print(c)
5     #如果调用函数试试的话，会发现函数会将d的所有键打印出来；
6     #也就是遍历的是d的键，而不是值。
```

做测试的时候，想要输出 (key:value) 的组合可以这样：

```
1 for c in dict:
2     print(c,':',dict[c])
```

或者：

```
1 for c in dict:
2     print(c,end=':'); print(dict[c])
```

4.10. Python 数据类型转换

Python 中数据类型的转换，只需要将数据类型作为函数名即可。

Function Name	Description
<code>int(x[,base])</code>	将 x 转换为一个整数
<code>float(x)</code>	将 x 转换为一个浮点数
<code>complex(real[,imag])</code>	创建一个复数
<code>str(x)</code>	将对象 x 转换为字符串
<code>repr(x)</code>	将对象 x 转换为表达式字符串
<code>eval(str)</code>	用来计算在字符串中的有效Python表达式，并返回一个对象
<code>tuple(s)</code>	将序列 s 转换为一个元组
<code>list(s)</code>	将序列 s 转换为一个列表
<code>set(s)</code>	将序列 s 转换为一个可变集合
<code>frozenset(s)</code>	将序列 s 转换为一个不可变集合
<code>dict(d)</code>	创建一个字典。 d 必须是一个包含 (key, value) 关系的序列 (sequence)。
<code>chr(x)</code>	将一个整数转换为一个字符
<code>ord(x)</code>	将一个字符转换为它的整数值
<code>hex(x)</code>	将一个整数转换为一个十六进制字符串
<code>oct(x)</code>	将一个整数转换为一个八进制字符串

`repr()` 函数将其对象 (Object) 转化为供解释器读取的形式:

```

1  >>>s = 'RUNOOB'
2  >>> repr(s)
3  "'RUNOOB'"
4  >>> dict = {'runoob': 'runoob.com', 'google': 'google.com'};
5  >>> repr(dict)
6  "'{'google': 'google.com', 'runoob': 'runoob.com'}'"

```

4.11. 数组,列表,矩阵之间的相互转化

Python 中提供的基本组合数据类型有集合、序列和字典，列表属于序列类型。数组 `array` 和矩阵 `mat` 的使用需要用到 `numpy` 库，它们可以相互便捷的转化。

```

1  from numpy import array, mat
2
3  #1.列表定义
4  a1 = [[1,2,3], [4,5,6]]
5  print('\n1.列表a1 :\n',a1)
6
7  #2.列表 -----> 数组
8  a2 = array(a1)
9  print('\n2.列表a1----->数组a2 :\n',a2)

```

```

10
11 #3.列表 ----> 矩阵
12 a3 = mat(a1)
13 print('\n3.列表a1---->矩阵a3 :\n',a3)
14
15 #4.数组 ----> 列表
16 a4 = a2.tolist()
17 print('\n4.数组a2---->列表a4:\n',a4)
18
19 #5.数组 ----> 矩阵
20 a5 = mat(a2)
21 print('\n5.数组a2---->矩阵a5:\n',a5)
22
23 #6.矩阵 ----> 列表
24 a6 = a3.tolist()
25 print('\n6.矩阵a3---->列表a6:\n',a6)
26
27 #7.矩阵 ----> 数组
28 a7 = array(a3)
29 print('\n7.矩阵a3---->数组a7:\n',a7)

```

输出结果：

```

1  1.列表a1 :
2  [[1, 2, 3], [4, 5, 6]]
3
4  2.列表a1---->数组a2 :
5  [[1 2 3]
6  [4 5 6]]
7
8  3.列表a1---->矩阵a3 :
9  [[1 2 3]
10 [4 5 6]]
11
12 4.数组a2---->列表a4:
13 [[1, 2, 3], [4, 5, 6]]
14
15 5.数组a2---->矩阵a5:
16 [[1 2 3]
17 [4 5 6]]
18
19 6.矩阵a3---->列表a6:
20 [[1, 2, 3], [4, 5, 6]]
21
22 7.矩阵a3---->数组a7:
23 [[1 2 3]
24 [4 5 6]]

```

5. Python 3 数据结构

5.1. 列表作堆栈

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放(后进先出)。用 `append()` 方法可以把一个元素添加到堆栈顶；用不指定索引的 `pop()` 方法可以把一个元素从堆栈顶释放出来。例如：

```
1 >>> stack = [3, 4, 5]
2 >>> stack.append(6)
3 >>> stack.append(7)
4 >>> stack
5 [3, 4, 5, 6, 7]
6 >>> stack.pop()
7 7
8 >>> stack
9 [3, 4, 5, 6]
10 >>> stack.pop()
11 6
12 >>> stack.pop()
13 5
14 >>> stack
15 [3, 4]
```

5.2. 列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。**每个列表推导式都在 `for` 之后跟一个表达式，然后有零到多个 `for` 或 `if` 子句。**返回结果是一个根据表达从其后的 `for` 和 `if` 上下文环境中生成出来的列表。

```
1 >>> [[x, x**2] for x in vec]                # 获得一个新的列表
2 [[2, 4], [4, 16], [6, 36]]
3 >>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
4 >>> [weapon.strip() for weapon in freshfruit] # 对序列里每一个元素逐个调用某方法
5 ['banana', 'loganberry', 'passion fruit']
6 >>> vec1 = [2, 4, 6]; vec2 = [4, 3, -9]
7 >>> [x*y for x in vec1 for y in vec2]        # 利用for语句从不同列表取值
8 [8, 6, -18, 16, 12, -36, 24, 18, -54]
9 >>> [3*x for x in vec if x > 3]             # 利用if语句添加限定条件
10 [12, 18]
11 >>> [vec1[i]*vec2[i] for i in range(len(vec1))]
12 [8, 12, -54]
13 >>> [str(round(355/113,i)) for i in range(1,6)] # 使用复杂表达式或嵌套函数
14 ['3.1', '3.14', '3.142', '3.1416', '3.14159']
15
16 [x*y for x in range(1,5) if x > 2 for y in range(1,4) if y < 3]
17 相当于：
18 for x in range(1,5):
19     if x > 2:
20         for y in range(1,4):
21             if y < 3:
22                 x*y
```

5.3. 遍历技巧

在字典 (List) 中遍历时，关键字和对应的值可以使用 `items()` 方法同时解读出来：

```

1 >>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
2 >>> for k, v in knights.items():
3 ...     print(k, v)
4 ...
5 gallahad the pure
6 robin the brave

```

在**序列 (seq)** 中遍历时，索引位置 and 对应值可以使用 `enumerate()` 函数同时得到：

```

1 >>> for i, v in enumerate(['tic', 'tac', 'toe']):
2 ...     print(i, v)
3 ...
4 0 tic
5 1 tac
6 2 toe

```

同时遍历两个或更多的序列，可以使用 `zip()` 组合：

```

1 >>> questions = ['name', 'quest', 'favorite color']
2 >>> answers = ['lancelot', 'the holy grail', 'blue']
3 >>> for q, a in zip(questions, answers):
4 ...     print('What is your {0}? It is {1}'.format(q, a))
5 ...
6 What is your name? It is lancelot.
7 What is your quest? It is the holy grail.
8 What is your favorite color? It is blue.

```

要反向遍历一个序列，首先指定这个序列，然后调用 `reversed()` 函数：

```

1 >>> for i in reversed(range(1, 10, 2)):
2 ...     print(i)
3 ...
4 9
5 7
6 5
7 3
8 1

```

要按顺序遍历一个序列，使用 `sorted()` 函数返回一个已排序的序列，并不修改原值：

```

1 >>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
2 >>> for f in sorted(set(basket)):
3 ...     print(f)
4 ...
5 apple
6 banana
7 orange
8 pear

```

6. Python 3 运算符

Python 支持以下类型的运算符：

- 算术运算符
- 比较(关系)运算符
- 赋值运算符
- 逻辑运算符
- 位运算符
- 成员运算符
- 身份运算符
- 运算符优先级

6.1. 算术运算符

Operator	Description
+	两个对象相加
-	两个对象相减
*	两个数相乘，或返回一个被重复若干次的字符串,列表或元组
/	两个数相除，返回一个浮点数
//	两个数相除，返回商的整数部分（向下取整）
%	两个数相除，返回除法的余数
**	幂运算

6.2. 比较(关系)运算符

Operator	Description
==	比较两个对象是否相等
!=	比较两个对象是否不等
>	比较前面一个对象是否大于后面一个对象
<	比较前面一个对象是否小于后面一个对象
>=	比较前面一个对象是否大于或等于后面一个对象
<=	比较前面一个对象是否小于或等于后面一个对象

6.3. 赋值运算符

Operator	Description
=	简单的赋值运算符
+=	加法赋值运算符，例： <code>c += a</code> 等效于 <code>c = c + a</code>
-=	减法赋值运算符，例： <code>c -= a</code> 等效于 <code>c = c - a</code>
*=	乘法赋值运算符，例： <code>c *= a</code> 等效于 <code>c = c * a</code>
/=	除法赋值运算符，例： <code>c /= a</code> 等效于 <code>c = c / a</code>
//=	取整除赋值运算符，例： <code>c //= a</code> 等效于 <code>c = c // a</code>
%=	取模赋值运算符，例： <code>c %= a</code> 等效于 <code>c = c % a</code>
**=	幂赋值运算符，例： <code>c **= a</code> 等效于 <code>c = c ** a</code>
:=	海象运算符，可在表达式内部为变量赋值。(Python 3.8 版本新增运算符)

没有 `:=` 之前：

```

1 | n = len(a)
2 | if n > 10:
3 |     print(f"List is too long({n} elements, expected <= 10)")

```

有了 `:=` 之后：

```

1 | if (n := len(a)) > 10:
2 |     print(f"List is too long ({n} elements, expected <= 10)")

```

第一行同时完成了：`len(a)` 的求值，将该值赋值给 `n`，再判断 `n` 是否大于10。这样就避免了多一次赋值给中间变量的步骤，或者避免调用 `len()` 两次。

另一个 `while` 循环控制的例子 (读取一个文件，当不为空则执行操作)，没有 `:=` 之前：

```

1 | while 1:
2 |     block = f.read(256)
3 |     if block != '':
4 |         process(block)

```

有了 `:=` 之后：

```

1 | while ( (block := f.read(256)) != '' ):
2 |     process(block)

```

6.4. 逻辑运算符

Operator	Logic Expression	Description
<code>and</code>	<code>x and y</code>	布尔 "与"：若 <code>x</code> 为 <code>False</code> ，则返回 <code>False</code> ；否则返回 <code>y</code> 的计算值。
<code>or</code>	<code>x or y</code>	布尔 "或"：若 <code>x</code> 是 <code>True</code> ，则返回 <code>x</code> 的值；否则返回 <code>y</code> 的计算值。
<code>not</code>	<code>not x</code>	布尔 "非"：若 <code>x</code> 为 <code>True</code> ，则返回 <code>False</code> ；否则返回 <code>True</code> 。

6.5. 位运算符

按位运算符是把数字看作二进制来进行计算的。Python 中的按位运算法则如下：

```
1 a = 0011 1100
2 b = 0000 1101
3 -----
4 a & b = 0000 1100
5 a | b = 0011 1101
6 a ^ b = 0011 0001
7 ~a = 1100 0011
```

Operator	Description
<code>&</code>	按位与运算符：参与运算的两个值, 如果两个相应位都为1, 则该位的结果为1；否则为0。
<code> </code>	按位或运算符：只要对应的两个值中的两个二进位有一个为1时，结果位就为1。
<code>^</code>	按位异或运算符：当两对应的二进位相异时，结果为1。
<code>~</code>	按位取反运算符：对数据的每个二进制位取反，即把1变为0，把0变为1。
<code><<</code>	左移动运算符：运算数的各二进位全部左移若干位，由 " <code><<</code> " 右边的数指定移动的位数，高位丢弃，低位补0。
<code>>></code>	右移动运算符：运算数的各二进位全部右移若干位，由 " <code>>></code> " 右边的数指定移动的位数，低位丢弃，高位补0。

实例：

```
1 #!/usr/bin/env python3
2 a = 60          # 60 = 0011 1100
3 b = 13          # 13 = 0000 1101
4 c = a & b;      # 12 = 0000 1100
5 print ("1 - c 的值为:", c)
6 c = a | b;      # 61 = 0011 1101
7 print ("2 - c 的值为:", c)
8 c = a ^ b;      # 49 = 0011 0001
9 print ("3 - c 的值为:", c)
10 c = ~a;         # -61 = 1100 0011
11 print ("4 - c 的值为:", c)
12 c = a << 2;     # 240 = 1111 0000
13 print ("5 - c 的值为:", c)
```



```
14 c = a >> 2;          # 15 = 0000 1111
15 print ("6 - c 的值为:", c)
16
17 a = 0b00111100      # 二进制赋值
18 bin(a)              # 输出二进制
19 oct(a)              # 输出八进制
20 hex(a)              # 输出十六进制
```

6.6. 成员运算符

Operator	Description
<code>in</code>	如果在指定的序列 (Sequence) 中找到值则返回 <code>True</code> ，否则返回 <code>False</code> 。
<code>not in</code>	如果在指定的序列 (Sequence) 中没有找到值则返回 <code>True</code> ，否则返回 <code>False</code> 。

6.7. 身份运算符

Operator	Description
<code>is</code>	判断两个标识符是不是引用自一个对象， <code>x is y</code> 类似于 <code>id(x) == id(y)</code>
<code>is not</code>	判断两个标识符是不是引用自不同对象， <code>x is not y</code> 类似于 <code>id(x) != id(y)</code>

注：`is` 与 `==` 的区别在于，`is` 用于判断两个变量引用对象是否为同一个，而 `==` 用于判断引用变量的值是否相等。

6.8. 运算符优先级

下表列出了从最高到最低优先级的所有运算符：

Operator	Description
<code>**</code>	幂运算 (最高优先级)
<code>~, +, -</code>	按位取反, 一元加号, 一元减号
<code>*, /, %, //</code>	乘法, 除法, 取余, 取整
<code>+, -</code>	加法, 减法
<code>>>, <<</code>	右移动运算符, 左移动运算符
<code>&</code>	按位与运算符
<code>^, </code>	按位异或运算符, 按位或运算符
<code><=, <, >, >=</code>	比较运算符 (小于等于, 小于, 大于, 大于等于)
<code>==, !=</code>	比较运算符 (等于, 不等于)
<code>=, %=, /=, //=, -=, +=, *=, **=</code>	赋值运算符
<code>is, is not</code>	身份运算符
<code>in, not in</code>	成员运算符
<code>not, and, or</code>	逻辑运算符

6.9. 无自增/自减运算

```

1  >>> b = 5
2  >>> a = 5
3  >>> id(a)
4  162334512
5  >>> id(b)
6  162334512
7  >>> a is b
8  True

```

Python 中变量是以内容为准而不是像C语言中以变量名为基准, 所以只要你的数字内容是相同的, 不管起什么名字, 这个变量的ID都是相同的, 这同时也说明了 Python 中一个变量可以以多个名称访问。这样的设计逻辑决定了 Python 中数字类型的值是不可变的, 因为如果 `a` 和 `b` 都是 5, 当你改变了 `a` 时, `b` 也会跟着变。因此, 正确的自增操作应该 `a = a + 1` 或者 `a += 1`, 当此 `a` 自增后, 通过 `id()` 观察可知 ID 值变化了, 即 `a` 已经是新值的名称。

```

1  >>> a=1000
2  >>> b=1000
3  >>> id(a);id(b)
4  2236612366224
5  2236617350384

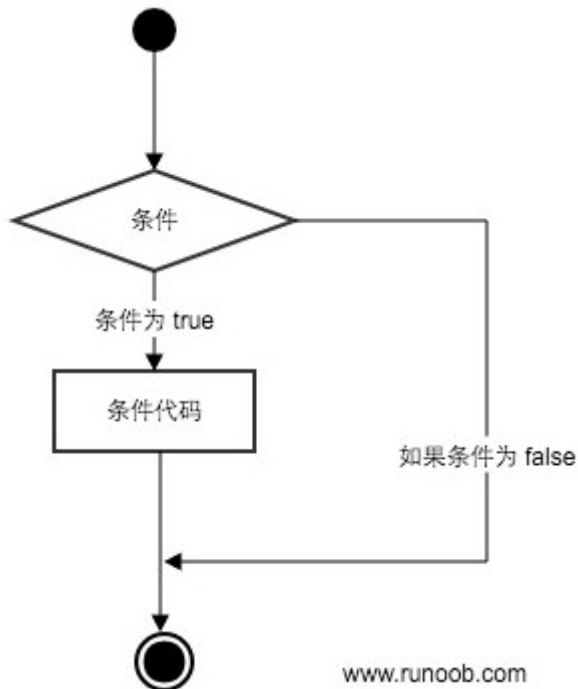
```

以上所说在脚本式编程环境中没有问题。但是在交互式环境中, 编译器会有一个小整数池的概念, 会把 (-5, 256) 间的数预先创建好, 而当 `a` 和 `b` 超过这个范围的时候, 两个变量就会指向不同的对象, 因此地址也会不一样。

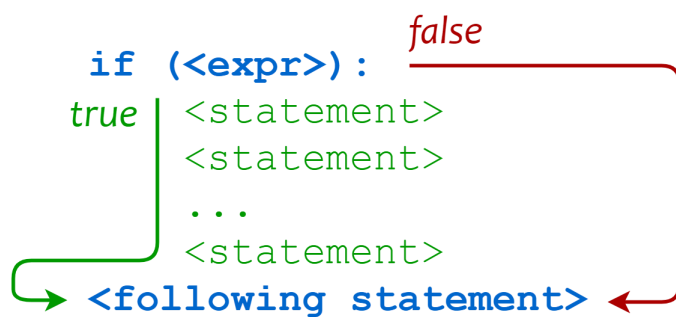
7. Python 3 控制语句

7.1. 条件控制

Python 条件语句是通过一条或多条语句的执行结果 (True 或 False) 来决定执行的代码块。执行过程图示：



代码执行过程：



7.1.1. if 语句

Python 中 `if` 语句的一般形式如下所示：

```
1 if condition_1:
2     statement_block_1
3 elif condition_2:
4     statement_block_2
5 else:
6     statement_block_3
```

- 若 "condition_1" 为 True, 将执行 "statement_block_1" 块语句
- 若 "condition_1" 为 False, 将判断 "condition_2"
- 若 "condition_2" 为 True, 将执行 "statement_block_2" 块语句
- 若 "condition_2" 为 False, 将执行 "statement_block_3" 块语句

注意：

- 1、每个条件后面要使用冒号(":")表示接下来是满足条件后要执行的语句块。
- 2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
- 3、在Python中没有 `switch - case` 语句。

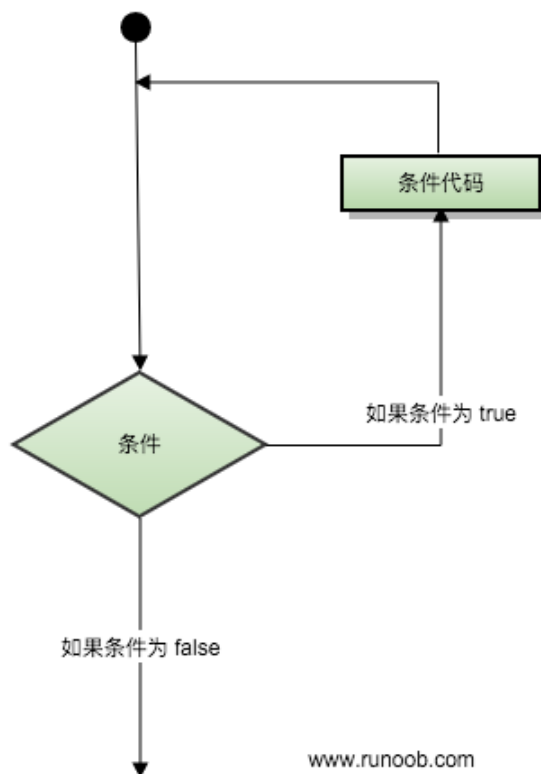
7.1.2. if 嵌套

在嵌套 `if` 语句中，可以把 `if...elif...else` 结构放在另外一个 `if...elif...else` 结构中。

```
1  if 表达式1:
2      语句
3      if 表达式2:
4          语句
5      elif 表达式3:
6          语句
7      else:
8          语句
9  elif 表达式4:
10     语句
11  else:
12     语句
```

7.2. 循环控制

Python 中的循环语句有 `for` 和 `while`，其控制结构图如下所示：



7.2.1. while 循环

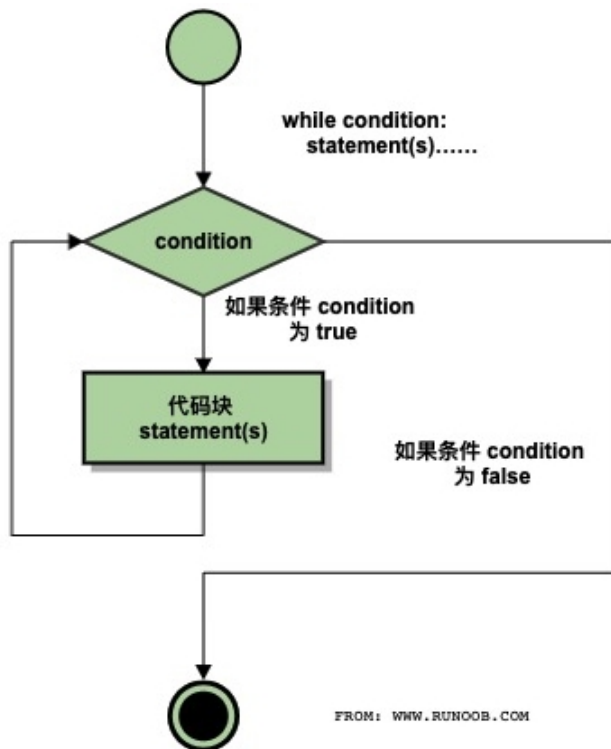
Python 中 `while` 循环语句的一般形式：

```

1  while <condition> :
2      <statement(s)>
3  或
4  while <expr>:
5      <statement(s)>
6  else:
7      <additional_statement(s)>
8  或
9  while <condition> : <statement(s)> # single-line style

```

执行流程图如下：



同样需要注意冒号和缩进。另外，在 Python 中**没有** `do...while` 循环。

7.2.2. for 循环

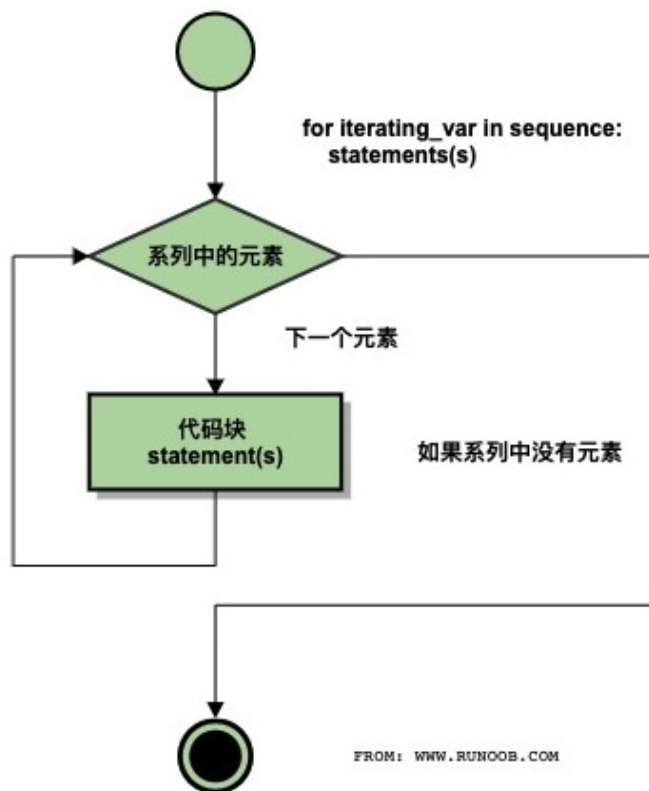
Python 中 `for` 循环可以遍历任何序列的项目，如一个列表或者一个字符串。`for` 循环的一般格式如下：

```

1  for <variable> in <sequence>:
2      <statements>
3  else:
4      <statements>

```

执行流程图如下：



如果需要遍历数字序列，可以使用内置 `range()` 函数生成数列；也可以使用 `range()` 指定区间的值并指定不同的步长 (可以为负数)；也可以结合 `range()` 和 `len()` 函数以遍历一个序列的索引：

```

1  >>> for i in range(5):
2  ...     print(i)
3  >>>
4  >>> for i in range(5,9):
5  ...     print(i)
6  >>>
7  >>> for i in range(0,10,3):
8  ...     print(i)
9  >>>
10 >>> for i in range(-10,-100,-30):
11 ...     print(i)
12 >>>
13 >>> a = ['Google', 'Baidu', 'Runoob', 'Taobao', 'QQ']
14 >>> for i in range(len(a)):
15 ...     print(i, a[i])

```

还可使用内置 `enumerate()` 函数进行遍历：

```

1  for index, item in enumerate(sequence):
2  ...     process(index, item)

```

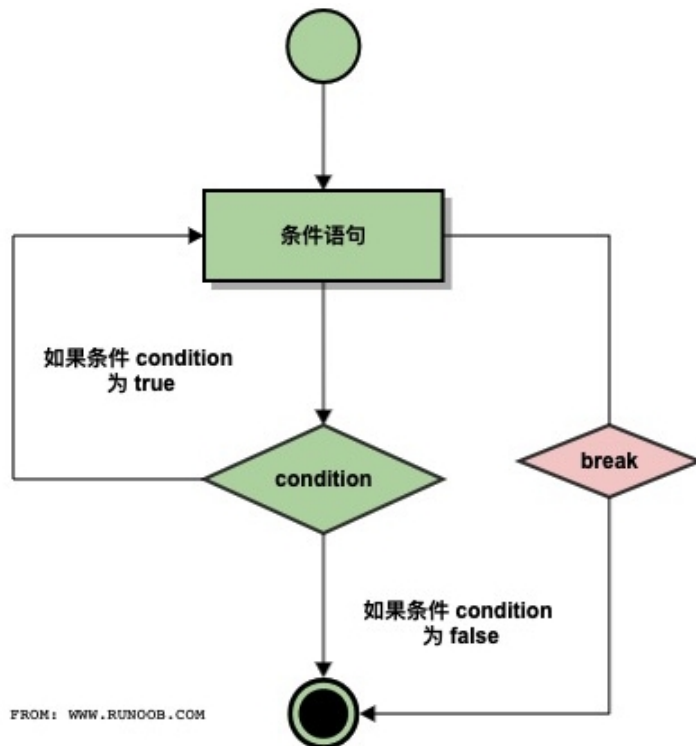
使用 `else` 的区别 (`while` 循环语句和 `for` 循环语句)：

- 若 `else` 语句和 `while` 循环语句一起使用，则当条件变为 `False` 时，则执行 `else` 语句。
- 若 `else` 语句和 `for` 循环语句一起使用，则 `else` 语句块只在 `for` 循环正常终止时执行！

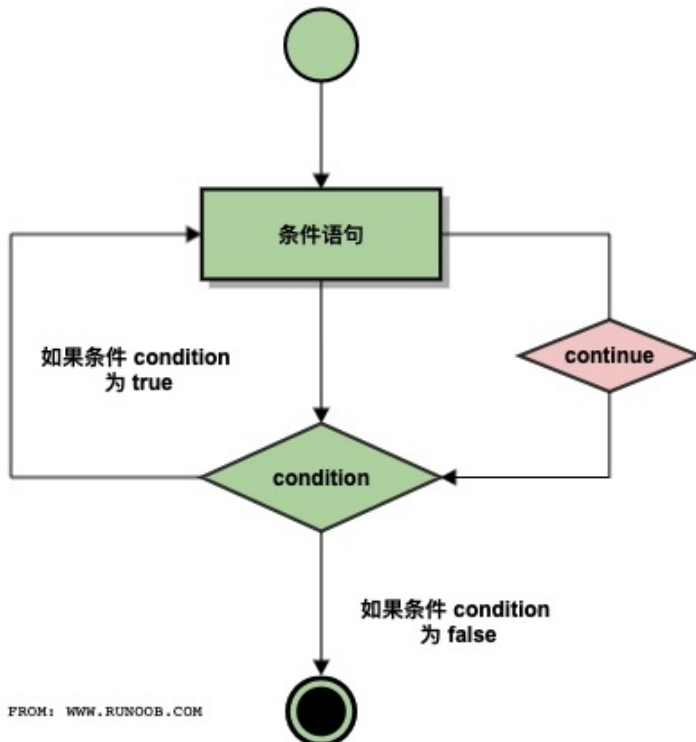
7.2.3. `pass`, `break` 和 `continue` 语句

pass语句是空语句，不做任何事情，是为了保持程序结构的完整性，一般用做占位语句。
break语句可以跳出 `for` 和 `while` 的循环体，若跳出循环体终止，任何对应的 `else` 循环块将不被执行。
continue语句被用来跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

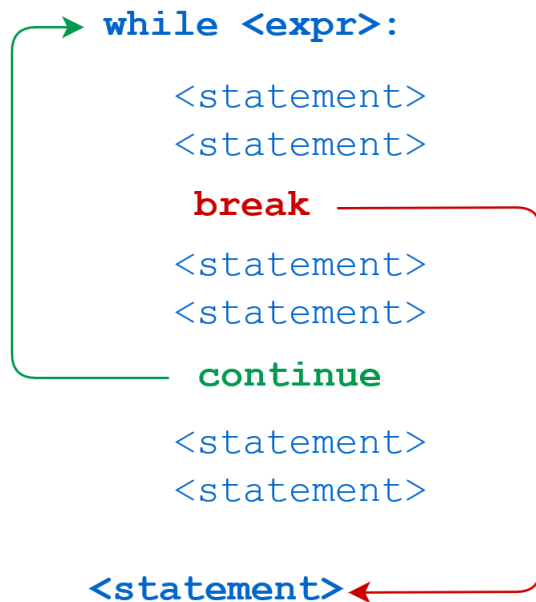
break 执行流程图：



continue 执行流程图：



代码执行过程：



8. Python 3 迭代器与生成器

8.1. 迭代器

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。迭代器有两个基本的方法：`iter()` 和 `next()`。

8.1.1. 迭代器的创建

字符串，列表或元组对象都可用于创建迭代器：

```
1 >>> list = [1,2,3,4]
2 >>> it = iter(list) # 创建迭代器对象
3 >>> print(next(it)) # 输出迭代器的下一个元素
4 1
5 >>> print(next(it))
6 2
```

迭代器对象可以使用常规 `for` 语句进行遍历：

```
1 #!/usr/bin/python3
2 list=[1,2,3,4]
3 it = iter(list) # 创建迭代器对象
4 for x in it:
5     print (x, end=" ")
```

也可以使用 `next()` 函数：


```

1  #!/usr/bin/python3
2  import sys      # 引入 sys 模块
3  list=[1,2,3,4]
4  it = iter(list) # 创建迭代器对象
5  while True:
6      try:
7          print(next(it))
8      except StopIteration:
9          sys.exit()

```

8.1.2. 类作为迭代器

把一个类作为一个迭代器使用需要在类中实现两个方法 `__iter__()` 与 `__next__()`。`__iter__()` 方法返回一个特殊的迭代器对象，这个迭代器对象实现了 `__next__()` 方法并通过 `StopIteration` 异常标识迭代的完成。`__next__()` 方法会返回下一个迭代器对象，在 `__next__()` 方法中我们可以设置在完成指定循环次数后触发 `StopIteration` 异常来结束迭代，防止出现无限循环的情况。

```

1  #!/usr/bin/env python3
2  class Fibonacci:
3      def __init__(self, count=10):
4          self._count = count
5
6      def __iter__(self):
7          self._a, self._b, self._i = 0, 1, 0
8          return self
9
10     def __next__(self):
11         if self._i < self._count:
12             self._i += 1
13             a = self._a
14             self._a, self._b = self._b, self._a + self._b
15             return a
16         else:
17             raise StopIteration
18
19     for i in Fibonacci(10):
20         print(i, end=" ")

```

8.2. 生成器

在 Python 中，使用了 `yield` 的函数被称为生成器 (generator)，跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。在调用生成器运行的过程中，每次遇到 `yield` 时函数会暂停并保存当前所有的运行信息，返回 `yield` 的值，并在下一次执行 `next()` 方法时从当前位置继续运行。

调用一个生成器函数，返回的是一个迭代器对象

```

1  #!/usr/bin/python3
2  import sys
3  def fibonacci(n): # 生成器函数 - 斐波那契
4      a, b, counter = 0, 1, 0
5      while True:
6          if (counter > n):
7              return

```

```

8         yield a
9         a, b = b, a + b
10        counter += 1
11
12    f = fibonacci(10) # f 是一个迭代器，由生成器返回生成
13    while True:
14        try:
15            print(next(f), end=" ")
16        except StopIteration:
17            sys.exit()

```

另一个例子：(文件读取。若直接对文件对象调用 `read()` 方法，会导致不可预测的内存占用。好的方法是利用固定长度的缓冲区来不断读取文件内容。而通过 `yield` 我们不再需要编写读文件的迭代类，即可轻松实现文件读取)

```

1    def read_file(fpath):
2        BLOCK_SIZE = 1024
3        with open(fpath, 'rb') as f:
4            while True:
5                block = f.read(BLOCK_SIZE)
6                if block:
7                    yield block
8                else:
9                    return

```

9. Python 3 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。函数能提高应用的模块性和代码的重复利用率。Python提供了许多**内建函数**，比如 `print()`，同时也可以自己创建函数，即**用户自定义函数**。

9.1. 定义函数

可以定义一个有自己想要功能的函数，以下是简单的规则：

- 函数代码块以 **def** 关键词开头，后接**函数标识符名称**和圆括号 **()**。
- 任何传入参数和自变量必须放在圆括号中间，圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串，用于存放函数说明。
- 函数内容以冒号起始，并且需要缩进。
- 结束函数 **return [表达式]**，选择性地返回一个值给调用方。不带表达式的相当于返回 `None`。

语法：

Python 定义函数使用 **def** 关键字，一般格式如下：

```

1    def 函数名(参数列表):
2        函数体

```

默认情况下，参数值和参数名称是按函数声明中定义的顺序匹配起来的。

9.2. 函数调用

函数定义：给了函数一个名称，指定了函数里包含的参数和代码块结构。这个函数的基本结构完成以后，可通过另一个函数调用执行，也可以直接从 Python 命令提示符执行。

函数返回值的注意事项: 不同于 C 语言，Python 函数可以返回多个值，多个值以元组的方式返回：

```
1  #!/usr/bin/env python3
2  def fun(a,b):
3      "返回多个值，结果以元组形式表示"
4      return a,b,a+b
5
6  print(fun(1,2))
```

9.3. 参数传递

Python 中一切都是对象，严格意义上不能说值传递还是引用传递，应该说传不可变对象和传可变对象：

- **不可变类型**：类似 C++ 的值传递，如整数, 字符串及元组。例如: `fun(a)` 传递的只是 `a` 的值，没有影响 `a` 对象本身，若在 `fun(a)` 内部修改 `a` 的值，则只是修改另一个复制的对象，而不会影响 `a` 本身。
- **可变类型**：类似 C++ 的引用传递，如列表, 字典及集合。例如: `fun(1a)` 则是将 `1a` 真正的传过去，修改后 `fun(1a)` 外部的 `1a` 也会受影响。

实例1 (传不可变对象)：

```
1  #!/usr/bin/python3
2  def ChangeInt(a):
3      a = 10
4
5  b = 2
6  ChangeInt(b)
7  print(b) # 结果仍是 2
```

实例2 (传可变对象):

```
1  #!/usr/bin/python3
2  # 可写函数说明
3  def changeme(mylist):
4      "修改传入的列表"
5      mylist.append([1,2,3,4])
6      print ("函数内取值：", mylist)
7      return
8
9  # 调用changeme函数
10 mylist = [10,20,30]
11 changeme(mylist)
12 print ("函数外取值：", mylist)
```

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

```
1  函数内取值： [10, 20, 30, [1, 2, 3, 4]]
2  函数外取值： [10, 20, 30, [1, 2, 3, 4]]
```

注意：函数也可以以一个函数为其参数：

```

1  #!/usr/bin/env python3
2  def hello () :
3      print ("Hello, world!")
4
5  def execute(f):
6      "执行一个没有参数的函数"
7      f()
8
9  execute(hello)

```

9.4. 参数类型

以下是调用函数时可使用的正式参数类型：

- 必需参数：须以正确的顺序传入函数，调用时的数量必须和声明时的一样。
- 关键字参数：函数调用使用关键字参数来确定传入的参数值，允许调用时参数的顺序与声明时不一致。
- 默认参数：调用函数时，若没有传递参数，则会使用默认参数。默认参数必须放在最后面。
- 不定长参数：可能需要一个函数能处理比当初声明时更多的参数，和上述参数不同，声明时不会命名。

实例1 (必须参数)：

```

1  #!/usr/bin/env python3
2  def printme(str):
3      "打印任何传入的字符串"
4      print(str)
5      return
6
7  # 调用printme()函数，不加参数会报错
8  printme()

```

实例2 (关键字参数)：

```

1  #!/usr/bin/env python3
2  def printinfo(name,age):
3      "打印任何传入的字符串"
4      print("名字: ", name)
5      print("年龄: ", age)
6      return
7
8  #调用printinfo()函数，不需要使用指定顺序
9  printinfo(age=50, name="runoob")

```

实例3 (默认参数)：

```

1  #!/usr/bin/env python3
2  def printinfo(name, age=35):
3      "打印任何传入的字符串"
4      print("名字: ", name)
5      print("年龄: ", age)
6      return
7
8  #调用printinfo()函数，若没有给定传递参数则使用默认参数
9  printinfo(age=50, name="runoob")
10 print ("-----")
11 printinfo(name="runoob")

```

不定长参数：

```

1  def functionname([formal_args,] *var_args_tuple, **var_args_dictionary):
2      "函数_文档字符串"
3      function_suite
4      return [expression]

```

加了一个星号 `*` 的参数会以元组 (tuple) 的形式导入，存放所有未命名的变量参数。

```

1  #!/usr/bin/env python3
2  def printinfo( arg1, *vartuple ):
3      "打印任何传入的参数"
4      print("输出: ")
5      print(arg1)
6      for var in vartuple:
7          print(var)
8      return
9
10 # 调用printinfo()函数
11 printinfo(70,60,50)

```

加了两个星号 `**` 的参数会以字典 (dict) 的形式导入，存放所有未命名的变量参数。

```

1  #!/usr/bin/env python3
2  def printinfo(arg1, **vardict):
3      "打印任何传入的参数"
4      print("输出: ")
5      print(arg1)
6      for key,value in vardict.items():
7          print(key,value)
8      return
9
10 # 调用printinfo 函数
11 printinfo(1, a=2,b=3)

```

声明函数时，参数中星号 `*` 可以单独出现，若单独出现星号 `*`，其后的参数必须用关键字传入。

```

1  >>> def f(a,b,*,c):
2      ...     return a+b+c
3      ...
4  >>> f(1,2,3)    # 报错
5  Traceback (most recent call last):
6      File "<stdin>", line 1, in <module>
7  TypeError: f() takes 2 positional arguments but 3 were given
8  >>> f(1,2,c=3) # 正常
9  6

```

9.5. 匿名函数 (lambda)

Python 使用 `lambda` 来创建匿名函数。所谓匿名，即不再使用 `def` 语句这样标准的形式定义一个函数。

- `lambda` 只是一个表达式，函数体比 `def` 简单很多。
- `lambda` 的主体是一个表达式，而不是一个代码块，仅仅能在 `lambda` 表达式中封装有限的逻辑进去。
- `lambda` 函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。
- `lambda` 函数不等同于 C 或 C++ 的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法：

`lambda` 函数的语法只包含一个语句：

```

1  lambda [arg1 [,arg2,...,argn]]:expression    # lambda函数也可以设定默认值

```

如下实例：

```

1  #!/usr/bin/python3
2  # 可写函数说明
3  sum = lambda arg1, arg2: arg1 + arg2
4  # 调用sum函数
5  print("相加后的值为 ： ", sum(10,20))
6  print("相加后的值为 ： ", sum(20,20))

```

匿名函数 `lambda` 常与 `map()` 函数一起使用，`map()` 函数会根据提供的函数对指定序列做映射：

```

1  map(function, iterable, ...)

```

第一个参数 `function` 以参数序列中的每一个元素调用该函数，返回包含每次 `function` 函数返回值的新列表。

```

1  >>>def square(x):    # 计算平方数
2      ...     return x ** 2
3      ...
4  >>> map(square, [1,2,3,4,5])    # 计算列表各个元素的平方
5  [1, 4, 9, 16, 25]
6  >>> map(lambda x: x ** 2, [1, 2, 3, 4, 5])    # 使用 lambda 匿名函数
7  [1, 4, 9, 16, 25]
8  >>> map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
9  [3, 7, 11, 15, 19]    # 提供了两个列表，对相同位置的列表数据进行相加

```

9.6. 强制位置参数

Python 3.8 中新增了一个函数形参语法 `/` 用来指明函数形参必须使用指定位置参数，不能使用关键字参数的形式。在以下的例子中，形参 `a` 和 `b` 必须使用指定位置参数，`c` 或 `d` 可以是位置形参或关键字形参，而 `e` 或 `f` 要求为关键字形参：

```
1 def f(a, b, /, c, d, *, e, f):
2     print(a, b, c, d, e, f)
```

以下使用方法是正确的：

```
1 f(10, 20, 30, d=40, e=50, f=60)
```

以下使用方法会发生错误：

```
1 f(10, b=20, c=30, d=40, e=50, f=60)    # b 不能使用关键字参数的形式
2 f(10, 20, 30, 40, 50, f=60)           # e 必须使用关键字参数的形式
```

10. Python 3 变量前加 * 或 ** 号

10.1. 变量前加 * 号可进行拆分

在列表、元组、字典变量前加 `*` 号，会将其拆分成一个一个的独立元素。不光是列表、元组、字典，由 `numpy` 生成的向量也可进行拆分。

```
1 >>> _list = [1, 3, 5, 2]
2 >>> _tuple = (1, 2, 4, 5)
3 >>> _dict = {'1': 'a', '2': 'b', '3': 'c'}
4 >>> print(_list, '=', *_list)
5 [1, 3, 5, 2] = 1 3 5 2
6 >>> print(_tuple, '=', *_tuple)
7 (1, 2, 4, 5) = 1 2 4 5
8 >>> print(_dict, '=', *_dict)
9 {'1': 'a', '2': 'b', '3': 'c'} = 1 2 3
```

此外，`*` 号也可以作用于高维的列表。例如拆分一个二维列表，其结果是两个一维列表：

```
1 >>> _list2 = [[1, 2, 3], [4, 5, 6]]
2 >>> print(*_list2)
3 [1, 2, 3] [4, 5, 6]
```

10.2. 函数传参中使用 * 或 **

函数的参数传递中使用 `*args` 和 `**kwargs`，这两个形参都接收若干个参数，通常我们将其称为参数组；

- `*args`：接收若干个位置参数，转换并存储于一个元组 (tuple) 变量 `args` 中；
- `**kwargs`：接收若干个关键字参数，转换并存储于一个字典 (dict) 变量 `kwargs` 中；
- 注意：位置参数 `*args` 一定要在关键字参数 `**kwargs` 前。

实例：

```

1  >>> def test(*args):
2  ...   print(args)
3  ...   return args
4  ...
5  >>> print(type(test(1,2,3,4)))
6  (1, 2, 3, 4)      # print(args)的结果, 其中 args = (1,2,3,4)
7  <class 'tuple'>   # print(type(args))的结果, test(1,2,3,4)返回的args是一个元组

```

10.3. 综合以上两点的实例

```

1  def add(*args) :
2  ...   print(type(args))
3  ...   for item in args :
4  ...       print(item)
5
6  _list = [1, 2, 4, 5]
7  add(_list) # 入参为1个列表 [1, 2, 4, 5]; 经*args后变为1个元组 args = ([1,2,4,5],) 仅包含1个元素
8  add(*_list) # 入参为4个元素 1, 2, 4, 5; 经*args后变为1个元组 args = (1,2,4,5) 包含4个元素

```

输出结果为：

```

1  <class 'tuple'>
2  [1, 2, 4, 5]
3  <class 'tuple'>
4  1
5  2
6  4
7  5

```

作用于二维列表的实例：

```

1  def add_plus(*args) :
2  ...   for item in args :
3  ...       print(item)
4
5  _list2 = [[1, 2, 3], [4, 5, 6]]
6  add_plus(_list2)
7  add_plus(*_list2)

```

输出结果为：

```

1  [[1, 2, 3], [4, 5, 6]]
2  [1, 2, 3]
3  [4, 5, 6]

```

10.4. 使用zip()函数进行压缩

Python 中有一个 `zip()` 函数功能与 `"*"` 号相反，该函数可将一个或多个可迭代对象进行包装压缩，返回的结果是一个 `'zip'` 类的迭代器。通俗的说：`zip()` 压缩可迭代对象，而 `"*"` 号解压可迭代对象。


```
1 用法： zip([iterable1, iterable2, ...])
2
3 说明： 创建一个聚合了来自每个可迭代对象中的元素的迭代器。返回一个元组的迭代器，其中的第 i 个
    元组包含来自每个参数序列或可迭代对象的第 i 个元素。当所输入可迭代对象中最短的一个被耗尽时，
    迭代器将停止迭代。当只有一个可迭代对象参数时，它将返回一个单元组的迭代器。若不带参数，它将返
    回一个空迭代器。
4
5 注意： zip()的结果为一个'zip'类，要经过 list() 之后才能显示出来。
```

实例1 (单迭代对象为参数)：

```
1  >>> x = [1, 2, 3]
2  >>> x = list(zip(x))
3  >>> print(x)
4  [(1,), (2,), (3,)]
```

实例2 (把两个列表转化为一个列表，以元组为该新列表的元素)：

```
1  >>> seq1 = ['one', 'two', 'three']
2  >>> seq2 = [1, 2, 3]
3  >>> list(zip(seq1,seq2))
4  [('one', 1), ('two', 2), ('three', 3)]
```

实例3 (把两个列表转化为一个列表，每个列表转换为一个元组)：

```
1  >>> zz = zip(seq1,seq2)
2  >>> list(zip(*zz))
3  [('one', 'two', 'three'), (1, 2, 3)]
```

实例4 (可利用 zip() 函数的特性可用来构建字典)：

```
1  >>> dict(zip(seq1,seq2))
2  {'one': 1, 'two': 2, 'three': 3}
```

实例5 (另一个构建字典的例子)：

```
1  >>> lst1 = ['food', 'drinks', 'sports']
2  >>> lst2 = [['hamburger', 'beer', 'football'], ['cheeseburger', 'wine', 'tennis']]
3  >>> [dict(zip(lst1, l)) for l in lst2]
4  [{'food': 'hamburger', 'drinks': 'beer', 'sports': 'football'}, {'food':
    'cheeseburger', 'drinks': 'wine', 'sports': 'tennis'}]
```

实例6 (应用于二维列表的例子)：

```
1  m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2  n = [[2, 2, 2], [3, 3, 3], [4, 4, 4]]
3
4  print('list(zip(m,n)):\n',list(zip(m,n)))
5  print("*zip(m, n):\n", *zip(m, n))
6  print("*zip(*zip(m, n)):\n", *zip(*zip(m, n)))
7
8  m2,n2 = zip(*zip(m, n))
9  print(m == list(m2) and n == list(n2))
```

输出结果：

```
1 list(zip(m,n)):  
2  [([1, 2, 3], [2, 2, 2]), ([4, 5, 6], [3, 3, 3]), ([7, 8, 9], [4, 4, 4])]  
3  *zip(m, n):  
4  ([1, 2, 3], [2, 2, 2]) ([4, 5, 6], [3, 3, 3]) ([7, 8, 9], [4, 4, 4])  
5  *zip(*zip(m, n)):  
6  ([1, 2, 3], [4, 5, 6], [7, 8, 9]) ([2, 2, 2], [3, 3, 3], [4, 4, 4])  
7  True
```

注意：

1. 可迭代对象才可以使用 "*" 号来拆分，或 `zip()` 函数来压缩；
2. 带 "*" 号变量并不是一个变量，而更应该称为参数，它是不能赋值给其他变量的，但可作为参数传递。

11. Python 3 命名空间和作用域

12. Python 3 模块

13. Python 3 输入和输出

14. Python 3 文件操作

15. Python 3 OS操作

16. Python 3 错误和异常

17. Python 中的浅拷贝与深拷贝

17.1. 赋值语句

```
1 a = 'abc'
2 b = a
3 print id(a)
4 print id(b)
5
6 # id(a):29283464
7 # id(b):29283464
```

通过简单的赋值语句，我们可以看到 `a`、`b` 其实是一个对象。对象赋值实际上是简单的对象引用，也就是说，当你创建了一个对象，然后把它赋值给另一个变量时，Python 并没有拷贝这个对象，而是拷贝了这个对象的引用。

17.2. 浅拷贝

序列 (Sequence) 类型的对象默认拷贝类型是浅拷贝，通过以下几种方式实施：

1. 完全切片操作，即 `[:]`；
2. 利用工厂函数，如 `list()`、`dict()` 等；
3. 使用 `copy` 模块中的 `copy()` 函数。

创建一个列表，然后分别用切片操作和工厂方法拷贝对象，然后使用 `id()` 内建函数来显示每个对象的标识符。

```
1 s = ['abc', ['def', 1]]
2 a = s[:]
3 b = list(s)
4 print([id(x) for x in (s,a,b)])
5 # [139780055330112, 139780053990464, 139780054532160]
```

可以看到创建了三个不同的列表对象。再对对象的每一个元素进行操作：

```
1 a[0] = 'a'
2 b[0] = 'b'
3 print(a,b)
4 # ['a', ['def', 1]] ['b', ['def', 1]]
5
6 a[1][1] = 0
7 print(a,b)
8 # ['a', ['def', 0]] ['b', ['def', 0]]
```

我们可以看到，当执行 `a[1][1] = 0` 时，`b[1][1]` 也跟着变为0。这是因为我们仅仅做了一个浅拷贝，对一个对象进行浅拷贝其实是新建了一个类型跟原对象一样，它的内容元素是原来对象元素的引用。换句话说，这个拷贝的对象是新的，但他的内容还是原来的，这就是浅拷贝。

```

1  #改变前
2  print([id(x) for x in a])
3  # [139780055253360, 139780055330304]
4  print([id(x) for x in b])
5  # [139780055253360, 139780055330304]
6
7  #改变后
8  print([id(x) for x in a])
9  # [139780054899056, 139780055330304]
10 print([id(x) for x in b])
11 # [139780055136176, 139780055330304]

```

但是我们看到 `a` 的第一个元素，即字符串被赋值后，并没有影响 `b` 的。这是因为在这个对象中，第一个字符串类型对象是不可变的，而第二个列表对象是可变的。正因为如此，当进行浅拷贝时，字符串被显式的拷贝，并创建了一个新的字符串对象，而列表元素只是把它的引用复制了，并不是他的成员。

17.3. 深拷贝

根据上面的例子，如果我们想要在改变 `a` 时不影响到 `b`，要得到一个完全拷贝或者说深拷贝 (即一个新的容器对象包含原有对象元素全新拷贝的引用)，就需要 `copy.deepcopy()` 函数。

```

1  from copy import deepcopy
2  s = ['abc', ['def', 1]]
3  a = deepcopy(s)
4  b = deepcopy(s)
5  print([id(x) for x in (s,a,b)])
6  # [139741157573888, 139741157596928, 139741157650240]
7  a[0] = 'a'
8  b[0] = 'b'
9  a[1][1] = 0
10 print(a,b)
11 # ['a', ['def', 0]] ['b', ['def', 1]]

```

18. Python 获取命令行参数

18.1. 利用 `sys.argv`

Python 中可以用 `sys` 的 `sys.argv` 来获取命令行参数：

```

1  sys.argv 是命令行参数列表。
2  len(sys.argv) 是命令行参数个数
3
4  注：sys.argv[0] 表示代码本身文件路径，所以参数从1开始

```

18.1.1. 实例1

创建 `test.py` 文件，代码如下：

```

1  #!/usr/bin/env python3
2  import sys
3  print ('参数个数为:', len(sys.argv), '个参数。')
4  print ('参数列表:', str(sys.argv))

```

执行以上代码，输出结果为：

```

1  $ python3 test.py arg1 arg2 arg3
2  参数个数为: 4 个参数。
3  参数列表: ['test.py', 'arg1', 'arg2', 'arg3']

```

18.1.2. 实例2

创建sample.py 文件，代码如下：

```

1  #!/usr/bin/env python
2  #_*_ coding:utf-8 *_
3  import sys
4
5  HELP = '''
6  This program prints files to the standard output.
7  Any number of files can be specified.
8  Options include:
9      --version : Prints the version number
10     --help    : Display this help
11  '''
12
13  def readfile(filename): #定义readfile函数，从文件中读出文件内容
14     '''Print a file to the standard output.'''
15     f = file(filename)
16     while True:
17         line = f.readline()
18         if len(line) == 0:
19             break
20         print line, # notice comma 分别输出每行内容
21     f.close()
22
23  # Script starts from here
24  print sys.argv
25
26  if len(sys.argv) < 2:
27      print 'No action specified.'
28      sys.exit()
29
30  if sys.argv[1].startswith('--'):
31      option = sys.argv[1][2:]
32      # fetch sys.argv[1] but without the first two characters
33      if option == 'version': #当命令行参数为-- version，显示版本号
34          print 'Version 1.2'
35      elif option == 'help': #当命令行参数为--help时，显示相关帮助内容
36          print HELP
37      else:
38          print 'Unknown option.'
39      sys.exit()
40  else:
41      for filename in sys.argv[1:]: #当参数为文件名时，传入readfile，读出其内容

```

在与sample.py同一目录下，新建1个记事本文件test.txt, 其内容为: `hello python!`。
验证sample.py，如下：

```

1 C:\Users\91135\Desktop>python sample.py
2 ['sample.py']
3 No action specified.
4
5 C:\Users\91135\Desktop>python sample.py --help
6 ['sample.py', '--help']
7 This program prints files to the standard output.
8 Any number of files can be specified.
9 Options include:
10 --version : Prints the version number
11 --help    : Display this help
12
13 C:\Users\91135\Desktop>python sample.py --version
14 ['sample.py', '--version']
15 Version 1.2
16
17 C:\Users\91135\Desktop>python sample.py --ok
18 ['sample.py', '--ok']
19 Unknown option.
20
21 C:\Users\91135\Desktop>python sample.py test.txt
22 ['sample.py', 'test.txt']
23 hello python!

```

18.2. 利用getopt模块

getopt 模块是专门处理命令行参数的模块，用于获取命令行选项和参数，即 `sys.argv`。命令行选项使得程序的参数更加灵活。支持短选项模式 (-) 和长选项模式 (--)。该模块提供了两个方法及一个异常处理来解析命令行参数。

18.2.1. getopt.getopt 方法

getopt.getopt 方法用于解析命令行参数列表，语法格式如下：

```
1 getopt.getopt(args, options[, long_options])
```

方法参数说明：

```

1 args      : 要解析的命令行参数列表。
2 options   : 以字符串的格式定义，后的冒号(:)表示该选项必须有附加参数，不带冒号表示该选项不
              附加参数。
3 long_options: 以列表的格式定义，后的等号(=)表示如果设置该选项则必须有附加参数，否则就不附加
              参数。
4
5 该方法返回值由两个元素组成：
6 第一个是 (option, value) 元组的列表。
7 第二个是参数列表，包含那些没有 '-' 或 '--' 的参数。

```

18.2.2. getopt.gnu_getopt 方法

另外一个方法是 'getopt.gnu_getopt'，这里不多做介绍。

18.2.3. 异常处理 except getopt.GetoptError

在没有找到参数列表，或选项的需要的参数为空时会触发该异常。

异常的参数是一个字符串，表示错误的原因。

属性 `msg` 和 `opt` 为相关选项的错误信息。

18.2.4. 实例

假定我们创建这样一个脚本，可以通过命令行向脚本文件传递两个文件名，同时我们通过另外一个选项查看脚本的使用。脚本使用方法如下：

```
1 | usage: test.py -i <inputfile> -o <outputfile>
```

创建test.py 文件，代码如下所示：

```
1 | #!/usr/bin/env python3
2 | import sys, getopt
3 |
4 | def main(argv):
5 |     inputfile = ''
6 |     outputfile = ''
7 |     try:
8 |         opts, args = getopt.getopt(argv,"hi:o:",["ifile=", "ofile="])
9 |     except getopt.GetoptError:
10 |         print ('test.py -i <inputfile> -o <outputfile>')
11 |         sys.exit(2)
12 |     for opt, arg in opts:
13 |         if opt == '-h':
14 |             print ('test.py -i <inputfile> -o <outputfile>')
15 |             sys.exit()
16 |         elif opt in ("-i", "--ifile"):
17 |             inputfile = arg
18 |         elif opt in ("-o", "--ofile"):
19 |             outputfile = arg
20 |         print ('Input File is: ', inputfile)
21 |         print ('Output File is: ', outputfile)
22 |
23 | if __name__ == "__main__":
24 |     main(sys.argv[1:])
```

```
1 | Note:
2 |     "hi:o:"          -> '-'型参数有: -h, -i(必须带附加参数), -o(必须带附加参数)
3 |     ["ifile=", "ofile="] -> '--'型参数有: --ifile(必须带附加参数), --ofile(必须带附加参数)
```

执行以上代码，输出结果为：

```
1 $ python3 test.py -h
2 usage: test.py -i <inputfile> -o <outputfile>
3 $ python3 test.py -i inputfile -o outputfile
4 Input File is: inputfile
5 Output File is: outputfile
```