

1. Python2 与 Python3 版本区别

- 1.1. `print` 函数
- 1.2. Unicode
- 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
- 4.5. 字符串 (String)
- 4.6. 列表 (List)

- 4. 7. 元祖 (Tuple)
- 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 运算符
- 6. Python 3 数字 (Number)
- 7. Python 3 字符串 (String)
- 8. Python 3 列表 (List)
- 9. Python 3 元组 (Tuple)
- 10. Python 3 字典 (Dictionary)
- 11. Python 3 集合 (Set)
- 12. Python 中的浅拷贝与深拷贝
 - 12. 1. 赋值语句
 - 12. 2. 浅拷贝
 - 12. 3. 深拷贝
- 13. Python 3 变量前加 * 或 ** 号
 - 13. 1. 变量前加 * 号可进行拆分
 - 13. 2. 函数传参中使用 * 或 **
 - 13. 3. 综合以上两点的实例
 - 13. 4. 使用 `zip()` 函数进行压缩
- 14. Python 获取命令行参数
 - 14. 1. 利用 `sys.argv`
 - 14. 1. 1. 实例1
 - 14. 1. 2. 实例2
 - 14. 2. 利用 `getopt` 模块
 - 14. 2. 1. `getopt.getopt` 方法
 - 14. 2. 2. `getopt.gnu_getopt` 方法
 - 14. 2. 3. 异常处理 `except getopt.GetoptError`
 - 14. 2. 4. 实例

Python3 基础教程

1. Python2 与 Python3 版本区别

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

Python2 有 ASCII `str()` 类型，`unicode()` 是单独的，不是 `byte` 类型。现在在 Python 3 中，我们最终有了 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

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`。

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 Module Name	New Module Name
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
repr	reprlib

StringIO 模块现在被合并到新的 io 模組内。
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

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；
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) 方法相互转化。

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

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

原 Python 2 中：

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

现改为只能用：

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

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
PYTHONPATH	Python搜索路径，默认import的模块都会从PYTHONPATH中寻找
PYTHONSTARTUP	Python启动后，先执行PYTHONSTARTUP环境变量指定的文件中的代码
PYTHONCASEOK	加入PYTHONCASEOK的环境变量，会使Python导入模块时不区分大小写
PYTHONHOME	另一种模块搜索路径，通常内嵌于PYTHONSTARTUP或PYTHONPATH目录中，使得两个模块库更容易切换

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：

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

即可直接运行脚本。

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，可以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，我们称之代码组。

像 `if`、`while`、`def` 和 `class` 这样的复合语句，首行以关键字开始，以冒号(:)结束，该行之后的一行或多行代码构成代码组。我们将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

Python3 仅保留了 `input()` 函数，其接收任意任性输入，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=' '`。这里表示我们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__':` 下的代码只有在第一种情况下 (即文件作为脚本直接执行) 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。在 Python 中, 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： <code>True</code> 或 <code>False</code> 。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()`。

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
```

4.5. 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。

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。

从后面索引：	-6	-5	-4	-3	-2	-1	
从前面索引：	0	1	2	3	4	5	
	+---+---+---+---+---+---+						
	a	b	c	d	e	f	
	+---+---+---+---+---+---+						
从前面截取：	:	1	2	3	4	5	:
从后面截取：	:	-5	-4	-3	-2	-1	:

加号 + 是字符串的连接符；星号 * 表示复制当前字符串，紧跟的数字为复制的次数。

```

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

```

Python 使用反斜杠 (\) 来转义；使用 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 中可使用三引号 (''' 或 ''') 可以指定一个多行字符串。

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。

列表元素的类型可以不相同，它支持数字、字符串，甚至可以包含列表 (即列表的嵌套)。

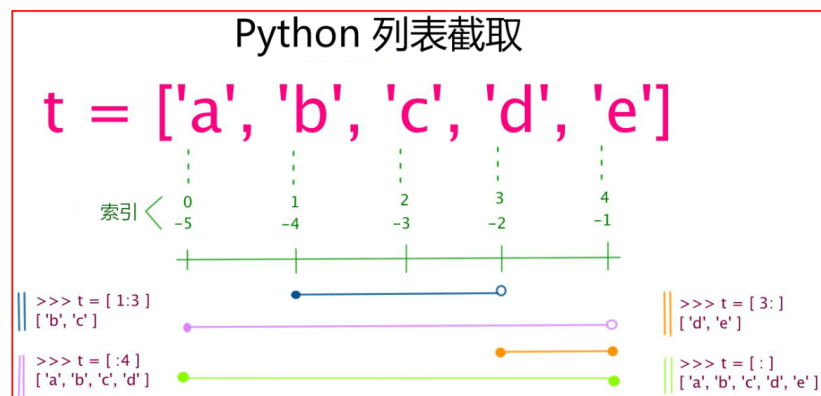
列表截取的语法格式如下：

```

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']
```



```
>>> 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 def reverseWords(input):
2     # 通过空格将字符串分隔符，把各个单词分隔为列表
3     inputWords = input.split(" ")
4     # 翻转字符串
5     # 假设列表 list = [1,2,3,4],
6     # list[0]=1, list[1]=2, 而-1表示最后一个元素 list[-1]=4 (与list[3]一样)
7     # inputWords[-1::-1] 有三个参数
8     # 第一个参数 -1 表示最后一个元素
9     # 第二个参数为空，表示移动到列表末尾
10    # 第三个参数为步长，-1 表示逆向
11    inputWords=inputWords[-1::-1]
12    # 重新组合字符串
13    output = ' '.join(inputWords)
14    return output
15
16 if __name__ == "__main__":
17     input = 'I like runoob'
18     rw = reverseWords(input)
19     print(rw)
```

输出结果为：

```
1 | runoob like I
```

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]
```

List 内置了有很多方法，例如 `append()`、`pop()` 等。

注意：

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

这两句话打印的内容其实是相似的：

```
1 2.23
2 [2.23]
```

但注意输出的结果是不同的类型：

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

4.7. 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。

```
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')

```

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。

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 >>>tup = (1, 2, 3, 4, 5, 6)
2 >>> print(tup[0])
3 1
4 >>> print(tup[1:5])
5 (2, 3, 4, 5)
6 >>> tup[0] = 11 # 修改元组元素的操作是非法的
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   TypeError: 'tuple' object does not support item assignment

```

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

```

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 parame = {value01,value02,...}
2 或者
3 set(value)

```

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

```

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 运算符

6. Python 3 数字 (Number)

7. Python 3 字符串 (String)

8. Python 3 列表 (List)

9. Python 3 元组 (Tuple)

10. Python 3 字典 (Dictionary)

11. Python 3 集合 (Set)

12. Python 中的浅拷贝与深拷贝

12.1. 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。

12.2. 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。

12.3. 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]]

```


13. Python 3 变量前加 * 或 ** 号

13.1. 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]
```

13.2. 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
```

13.3. 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]

```

13.4. 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, 2), (2, 2, 2), (3, 3, 3), (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。

14. Python 获取命令行参数

14.1. 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
```

14.1.1. 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']
```

14.1.2. 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
42             readfile(filename)

```

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!

```

14.2. 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。

14.2.1. 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。
```

14.2.2. getopt.gnu_getopt 方法

另外一个方法是 '`getopt.gnu_getopt`'，这里不多做介绍。

14.2.3. 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。

14.2.4. 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

```