

Python Tutorial

Shuai Wang

USTC, April 11, 2019

[Copyright \(c\) 2019 Shuai Wang. All rights reserved.](#)

*Shuai Wang copyrights this specification. No part of this specification may be reproduced in any form or means,
without the prior written consent of Shuai Wang.*

0. Quick Start

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very

readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
# Prints "[1, 1, 2, 3, 6, 8, 10]"
```

预备知识

Python versions and Installation

There are currently two different supported versions of Python, 2.7 and 3.5. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.5 and vice versa. For this class all code will use Python 3.5.

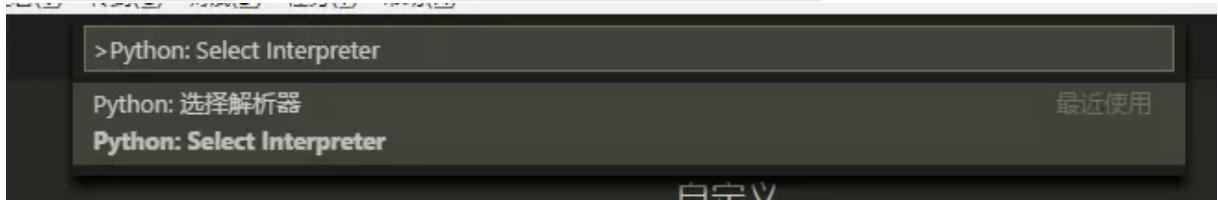
You can check your Python version at the command line by running

```
python --version.
```

使用VSCode 调试Python

- 首先，下载并安装VS Code：[下载地址](#)
 - 安装完成后，打开软件会自动提示你安装一些重要插件，如中文语言包，Git等，这里可以直接选择安装Python插件(也可以使用Ctrl+Shift+X可以打开扩展商店然后输入Python搜索)。可以使用微软的插件
 - 点击 install - reload
- 用VS打开项目文件夹
 - 首先，创建一个空文件夹"hello"，然后使用VS Code打开它。通过VS Code打开文件夹，该文件夹就变成了你的"工作区"。设定解释器后，VS Code在 `.vscode/settings.json` 中存储该工作区的特殊配置，与用户的全局设定相分开。
- 选取Python解释器

- 使用 **Ctrl+Shift+P** 打开命令板，输入 **Python: Select Interpreter** 进行搜索。



- 接下来会显示VS Code所能找到的全部解释器，选择你需要的那个就好。如果没找到你需要的那个，参考 [Python环境变量配置](#)。

• 创建Hello World

- 在Hello文件夹下新建文件，命名为hello.py
- 编写如下程序，可以自动补全，编写完成后按 **ctrl+s** 保存。

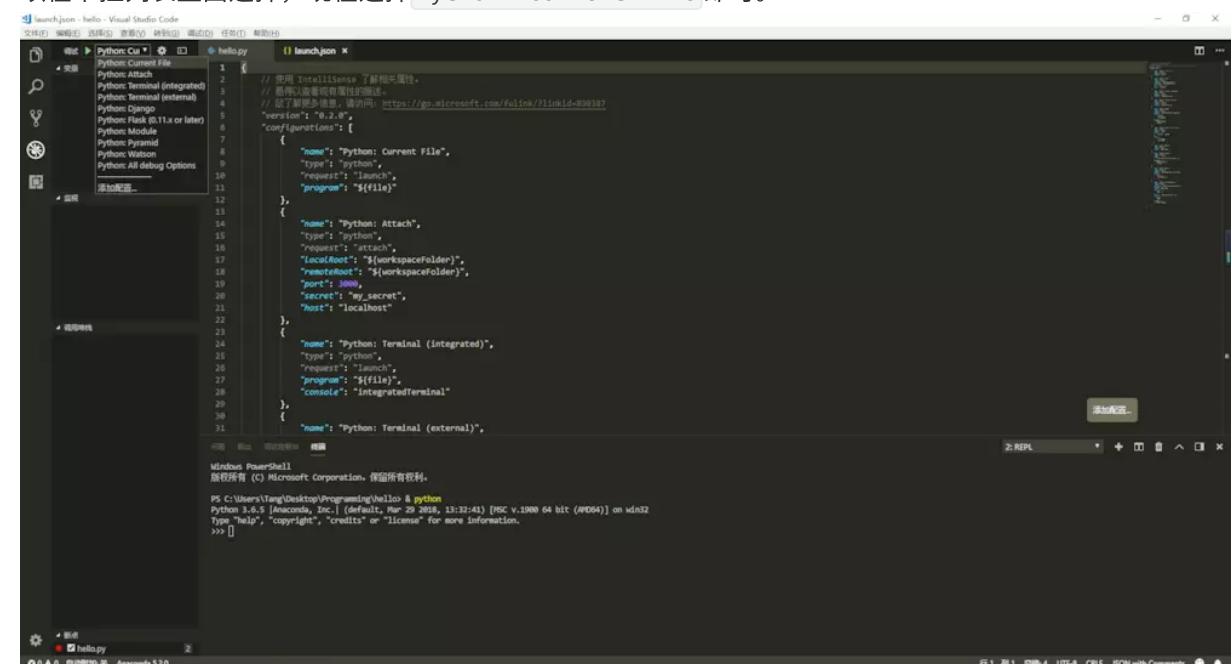
```
msg = "Hello World"
print(msg)
```

• 运行Hello World

- 在空白处右键选择**在终端运行Python文件**，就可以看到运行结果了。
- 此外，VS Code中还有一些运行Python代码的方式：
- 选择一行或者多行，使用 **Shift+Enter** 或者右键选择**在Python终端中运行选定内容/行**运行一部分代码。
- 编辑完代码按F5即可运行。初次运行会让你选环境，选择python即可。
 - 默认按F5后需要再按一次F5程序才会运行，如果要按F5马上运行需要将launch.json文件的 "stopOnEntry": true, 改为 "stopOnEntry": false。

• 配置及运行调试器

- 首先，把光标移到第二行然后按 **F9**，就可以设置一个断点。同样，也可以在行号左边双击设置。
- 接下来，在侧边栏打开Debug视图（蜘蛛图标）或 **ctrl+shift+D**
- 然后点击配置按钮，选择Python，然后Python插件会自动创建包含一系列配置的 **launch.json** 文件，可以在下拉列表里面选择，现在选择 **Python: Current File** 即可。



- 为了让调试器在自动在程序开始时停在第一行，这个功能顾名思义，意思就是在进入程序的时候就暂停执行，相当于在程序的第一行放一个断点。打开这个功能非常方便，只需要在 `lanuch.json` 中加入下图中红线那一行就可以了。这个功能是默认禁止的，所以删掉这一行或者把 `true` 改成 `false` 都可以起到禁止的效果。, 添加一条配置`stopOnEntry": true`

```
{
  "name": "Python: Current File",
  "type": "python",
  "request": "launch",
  "program": "${file}",
  "stopOnEntry": true
},
```

在编辑器中跳转回 `hello.py`，点击绿色箭头或者按 `F5` 启动调试器。调试器会停留在文件的第一行。

- 调试程序, 调试工具栏出现在页面上方, 从左到右功能分别是:
 - 运行 (`F5`) , 跳过 (`F10`) , 跳入 (`F11`) , 跳出 (`Shift+F11`) , 重新开始 (`Ctrl+Shift+F5`) , 以及停止 (`Shift+F5`) 。
 - 左侧分别为变量栏, 用来显示各种变量;
 - WATCH栏用来监控变量值变化, 可以通过选中某一变量, 然后右键选择: `Debug: add to watch`
- 编辑断点: 在断点上右键, 可以选择编辑断点, 比如使用`log message`选项, 可以添加一些输出信息 (输出变量用`{}`括起来) , 而不用终止或修改程序, 程序输出会在`Debug console`出现
- 编辑断点-expression: 可以写入一个条件表达式, 程序仅当此表达式为真时此断点被触发, 多用于循环语句中, 在某一循环处触发断点, 如: `name=="Testuser"`
- BREAKINGPOINTS栏选中 Rasied Exceptions, 程序会在发生异常处中断, 可手动设置一个异常进行尝试:
`raise ValueError()`

插件

`pylint`

配置`flake8`

安装`flake8`之后写代码的时候编辑器就会提示哪里出错, 代码格式不规范也会提示

1. 打开命令行
2. 输入 "`pip install flake8`"
3. 安装`flake8`成功后, 打开VScode, 文件->首选项->用户设置, 在`settings.json`文件中输入
`"python.linting.flake8Enabled": true`

```
1 // 通过将设置放入设置文件中来覆盖设置。
2 // See http://go.microsoft.com/fwlink/?LinkId=808995 for more information about the settings file.
3 {
4     // 编辑器配置
5     // 控制字体系列。
6     "editor.fontFamily": "Consolas, 'Courier New', monospace",
7
8     // 控制字体大小。
9     "editor.fontSize": 14,
10    // 控制行高。
11    "editor.lineHeight": 0,
12
13    // 控制行号的可见性
14    "editor.lineNumbers": true,
15
16
17 }
```

```
1 // 将设置放入此文件中以覆盖默认设置
2 {
3
4     // Whether to lint Python files using flake8
5     "python.linting.flake8Enabled": true,
6
7     // Provider for formatting. Possible options include
8     "python.formatting.provider": "yapf",
9
10    "editor.fontSize": 18,
11    "markdown.styles": [
12        "D:/Microsoft VS Code/css/markdown1.css"
13    ]
14
15 }
16 }
```

配置yapf

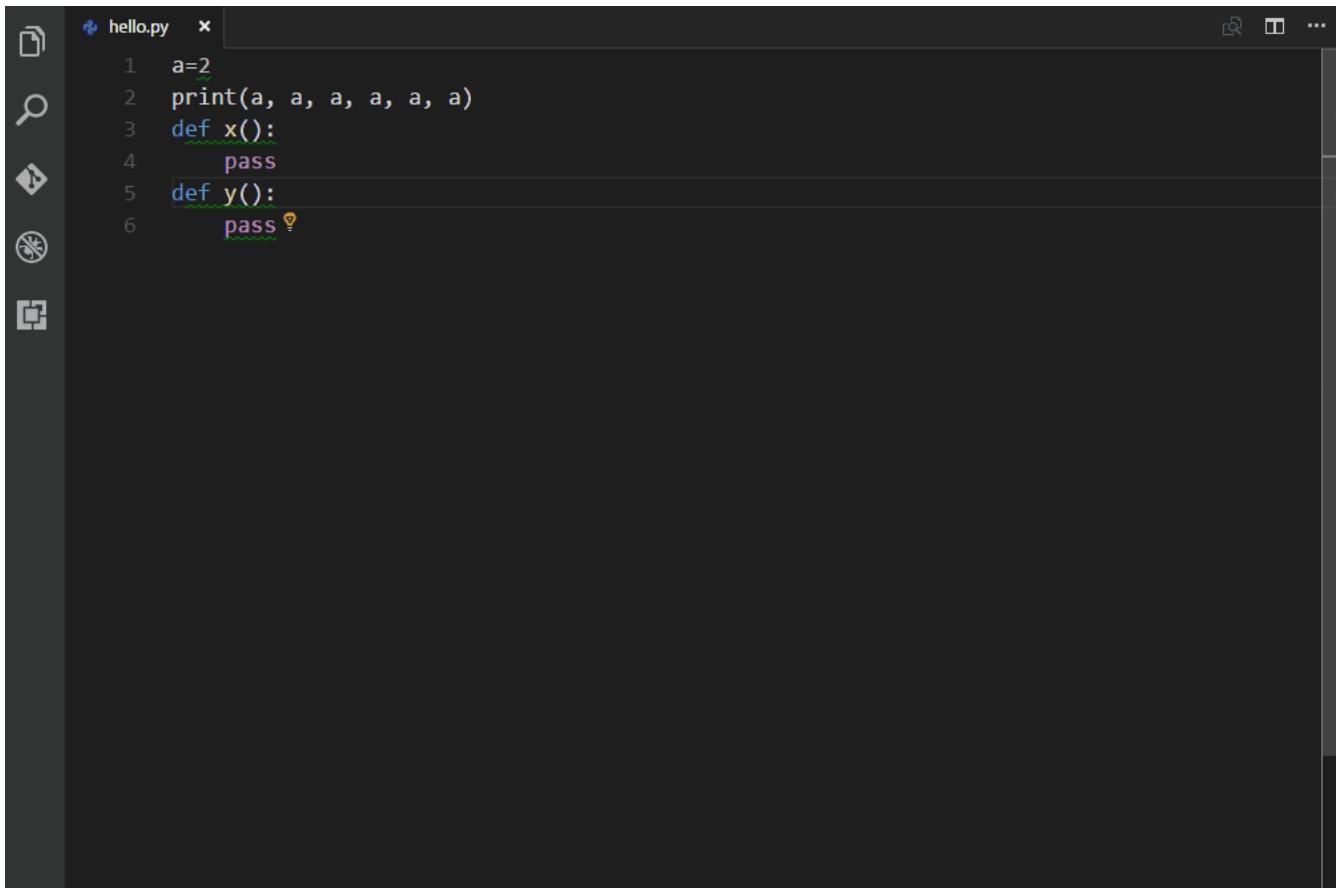
安装yapf之后在VSCode中按Alt+Shift+F即可自动格式化代码

1. 打开命令行
2. 输入 "pip install yapf"
3. 安装yapf成功后，打开VSCode，文件->首选项->用户设置，在settings.json文件中输入"python.formatting.provider": "yapf"

```
1 // 通过将设置放入设置文件中来覆盖设置。
2 // See http://go.microsoft.com/fwlink/?LinkId=808995 for more information about the settings file.
3 {
4     // 编辑器配置
5     // 控制字体系列。
6     "editor.fontFamily": "Consolas, 'Courier New', monospace",
7
8     // 控制字体大小。
9     "editor.fontSize": 14,
10    // 控制行高。
11    "editor.lineHeight": 0,
12
13    // 控制行号的可见性
14    "editor.lineNumbers": true,
15
16
17 }
```

```
1 // 将设置放入此文件中以覆盖默认设置
2 {
3
4     // Whether to lint Python files using flake8
5     "python.linting.flake8Enabled": true,
6
7     // Provider for formatting. Possible options include
8     "python.formatting.provider": "yapf",
9
10    "editor.fontSize": 18,
11    "markdown.styles": [
12        "D:/Microsoft VS Code/css/markdown1.css"
13    ]
14
15 }
16 }
```

yapf配置.png

A screenshot of the Visual Studio Code interface. On the left is the sidebar with icons for file, search, and other tools. The main area shows a Python file named 'hello.py' with the following code:

```
1 a=2
2 print(a, a, a, a, a, a)
3 def x():
4     pass
5 def y():
6     pass
```

The code is styled with green underlines and squiggly lines, indicating syntax errors or style violations. The status bar at the bottom right shows 'yapf'.

yapf效果图.gif

几个小技巧

1. 查看函数或者类的定义

Ctrl+鼠标左键点击函数名或者类名即可跳转到定义处，在函数名或者类名上按F12也可以实现同样功能

2. 更改变量名

在变量名上按F2即可实现重命名变量

3. python断点调试

在行号的左边点击即可设置断点，在左边的调试界面可以查看变量的变化

4. 隐藏菜单栏

这个属于个人习惯，如果你也感觉菜单栏很碍眼，可以点击查看->切换菜单栏，即可隐藏菜单栏。需要菜单栏的时候按Alt键即可查看

5. 设置快捷键

文件->首选项->键盘快捷方式，将需要修改的快捷键的整个大括号里面的内容复制到右边keybindings.json文件中，然后修改“key”的值为你需要的快捷键即可。我这边只修改了复制一行和删除一行的快捷键。

A screenshot of the Visual Studio Code interface showing two open files: '默认键盘快捷方式' and 'keybindings.json'. The 'keybindings.json' file contains the following JSON code:

```
1 // 通过将键绑定放入键绑定文件中
2 [
3     { "key": "shift+escape", },
4     { "key": "escape", },
5     { "key": "shift+escape", },
6     { "key": "escape", },
7     { "key": "ctrl+end", }
```

The 'keybindings.json' file also contains:

```
1 // 将键绑定放入此文件中以覆盖默认值
2 [
3     { "key": "ctrl+y", "command": "editor.action.deleteLines", "when": "editorTextFocus" },
4     { "key": "ctrl+d", "command": "editor.action.copyLinesDownAction", "when": "editorTextFocus" }
7 ]
```

快捷键设置.png

更新

1. 2016-8-25 更新
推介两个插件
2. Guides，缩进线插件，让代码看起来更清晰
3. vscode-todo，使VSCode支持TODO的插件

Basic data types

Like most languages, Python has a number of basic types including integers, floats, booleans, and strings. These data types behave in ways that are familiar from other programming languages.

Numbers: Integers and floats work as you would expect from other languages:

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)        # Prints "3"
print(x + 1)    # Addition; prints "4"
print(x - 1)    # Subtraction; prints "2"
print(x * 2)    # Multiplication; prints "6"
print(x ** 2)   # Exponentiation; prints "9"
x += 1
print(x)        # Prints "4"
x *= 2
print(x)        # Prints "6"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

Note that unlike many languages, Python does not have unary increment (`x++`) or decrement (`x--`) operators.

Python also has built-in types for complex numbers; you can find all of the details [in the documentation](#).

Booleans: Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True" (the same is False)
```

Strings: Python has great support for strings:

```

hello = 'hello'      # String literals can use single quotes
world = "world"      # or double quotes; it does not matter.
print(hello)         # Prints "hello"
print(len(hello))   # String length; prints "5"
hw = hello + ' ' + world # String concatenation
print(hw)            # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string formatting
print(hw12)          # prints "hello world 12"

```

String objects have a bunch of useful methods; for example:

```

s = "hello"
print(s.capitalize()) # Capitalize a string; prints "Hello"
print(s.upper())     # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))    # Right-justify a string, padding with spaces; prints "  hello"
print(s.center(7))   # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)')) # Replace all instances of one substring with another;
                             # prints "he(ell)(ell)o"
print(' world '.strip()) # Strip leading and trailing whitespace; prints "world"

```

You can find a list of all string methods [in the documentation](#).

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists

A list is the Python equivalent of an array, but is **resizeable** and **can contain elements of different types**:

Note: There is no array data type in Python.

```

xs = [3, 1, 2]      # Create a list
print(xs, xs[2])   # Prints "[3, 1, 2] 2"
print(xs[-1])       # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'        # Lists can contain elements of different types
print(xs)            # Prints "[3, 1, 'foo']"
xs.append('bar')     # Add a new element to the end of the list
print(xs)            # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()         # Remove and return the last element of the list
print(x, xs)         # Prints "bar [3, 1, 'foo']"

```

As usual, you can find all the gory details about lists [in the documentation](#).

Slicing:

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as *slicing*:

```

nums = list(range(5))      # range is a built-in function that creates a list of integers
print(nums)                # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])           # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])             # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])             # Get a slice from the start to index 2 (exclusive); prints "[0,
1]"
print(nums[:])              # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])            # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]           # Assign a new sublist to a slice
print(nums)                # Prints "[0, 1, 8, 9, 4]"

```

We will see slicing again in the context of numpy arrays.

Loops:

You can loop over the elements of a list like this:

```

animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.

```

Access index:

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

```

animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line

```

List comprehensions:

When programming, frequently we want to transform one type of data into another.

As a simple example, consider the following code that computes square numbers:

```

nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)  # Prints [0, 1, 4, 9, 16]

```

You can make this code simpler using a **list comprehension**:

```

nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)  # Prints [0, 1, 4, 9, 16]

```

List comprehensions can also contain conditions:

```

nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares) # Prints "[0, 4, 16]"

```

Dictionaries

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

```

d = {'cat': 'cute', 'dog': 'furry'} # Create a new dictionary with some data; The colon : is necessary
print(d['cat'])      # Get an entry from a dictionary; prints "cute"
print('cat' in d)    # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'    # Add an entry in a dictionary, not always at the end of the dic
print(d['fish'])     # Prints "wet"
# print(d['monkey']) # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A')) # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))   # Get an element with a default; prints "wet"
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"

```

You can find all you need to know about dictionaries [in the documentation](#).

Loops: It is easy to iterate over the keys in a dictionary:

```

d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"

```

If you want access to keys and their corresponding values, use the `items` method:

```

d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"

```

Dictionary comprehensions:

These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```

nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square) # Prints "{0: 0, 2: 4, 4: 16}"

```

Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

```

animals = {'cat', 'dog'}
print('cat' in animals)    # Check if an element is in a set; prints "True"
print('fish' in animals)   # prints "False"
animals.add('fish')        # Add an element to a set
print('fish' in animals)   # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"
animals.add('cat')         # Adding an element that is already in the set does nothing --
Unordered
print(len(animals))       # Prints "3"
animals.remove('cat')      # Remove an element from a set
print(len(animals))       # Prints "2"

```

As usual, everything you want to know about sets can be found [in the documentation](#).

Loops:

Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```

animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"

```

Set comprehensions:

Like lists and dictionaries, we can easily construct sets using set comprehensions:

```

from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums) # Prints "{0, 1, 2, 3, 4, 5}"

```

Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```

d = {(x, x + 1): x for x in range(10)} # Create a dictionary with tuple keys
t = (5, 6)      # Create a tuple
print(type(t))  # Prints "<class 'tuple'>"
print(d[t])     # Prints "5"
print(d[(1, 2)]) # Prints "1"

```

[The documentation](#) has more information about tuples.

Functions

Python functions are defined using the `def` keyword. For example:

```

def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"

```

We will often define functions to take optional keyword arguments, like this:

```

def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True) # Prints "HELLO, FRED!"

```

There is a lot more information about Python functions [in the documentation](#).

Classes

The syntax for defining classes in Python is straightforward:

```

class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred') # Construct an instance of the Greeter class
g.greet()           # Call an instance method; prints "Hello, Fred"
g.greet(loud=True) # Call an instance method; prints "HELLO, FRED!"

```

You can read a lot more about Python classes [in the documentation](#).

1. Beginner Level

1.1 Python Version and Installation

Python的3.0版本，常被称为Python 3000，或简称Py3k。相对于Python的早期版本，这是一个较大的升级。为了不带入过多的累赘，Python 3.0在设计的时候没有考虑向下兼容。

查看python版本

```
python -V
```

Anaconda

1.1 什么是 Anaconda?

Anaconda是专注于数据分析的Python发行版本，包含了conda、Python等190多个科学包及其依赖项。作为好奇宝宝的你是不是发现了一个新名词 conda，那么你一定会问 conda 又是什么呢？

1.2 什么是 conda ?

conda 是开源包（packages）和虚拟环境（environment）的管理系统。

- **packages 管理：** 可以使用 conda 来安装、更新、卸载工具包，并且它更关注于数据科学相关的工具包。在安装 anaconda 时就预先集成了像 Numpy、Scipy、pandas、Scikit-learn 这些在数据分析中常用的包。另外值得一提的是，conda 并不仅仅管理Python的工具包，它也能安装非python的包。比如在新版的 Anaconda 中就可以安装R语言的集成开发环境 Rstudio。
- **虚拟环境管理：** 在conda中可以建立多个虚拟环境，用于隔离不同项目所需的不同版本的工具包，以防止版本上的冲突。对纠结于 Python 版本的同学们，我们也可以建立 Python2 和 Python3 两个环境，来分别运行不同版本的 Python 代码。

二、如何安装Anaconda?

可以从这里下载 Anaconda 的安装程序以及查看安装说明。无论是 Windows、Linux 还是 MAC 的 OSX 系统，都可以找到对应的安装软件。如果你的电脑是64位则尽量选64位版本。至于 Python 的版本是 2.7 还是 3.x，这里推荐你使用 Python3，因为 Python2 终将停止维护。可能目前市面上大多数教程使用的都是 Python2，这也不用着急，因为在 Anaconda 中可以同时管理两个 Python 版本的环境。

根据提示进行安装，完成后你大概会惊讶地发现电脑中多了好多应用，不用担心，我们一项项来看：

- **Anaconda Navigator :** 用于管理工具包和环境的图形用户界面，后续涉及的众多管理命令也可以在 Navigator 中手工实现。
- **Jupyter notebook :** 基于web的交互式计算环境，可以编辑易于人们阅读的文档，用于展示数据分析的过程。
- **qtconsole :** 一个可执行 IPython 的仿终端图形界面程序，相比 Python Shell 界面，qtconsole 可以直接显示代码生成的图形，实现多行代码输入执行，以及内置许多有用的功能和函数。
- **spyder :** 一个使用Python语言、跨平台的、科学运算集成开发环境。

安装完成后，我们还需要对所有工具包进行升级，以避免可能发生的错误。打开你电脑的终端，在命令行中输入：

```
conda upgrade --all
```

在终端询问是否安装如下升级版本时，输入 y。

有的情况下，你可能会遇到找不到 conda 命令的错误提示，这很可能是环境路径设置的问题，需要添加 conda 环境变量：export PATH=xxx/anaconda/bin:\$PATH, 其中 xxx 替换成 anaconda 的安装路径。

至此，安装完成，下面让我们看一下如何用 Anaconda 管理工具包和环境。

三、如何管理 Python 包？

安装一个 package：

```
conda install package_name
```

这里 package_name 是需要安装包的名称。你也可以同时安装多个包，比如同时安装 numpy、scipy 和 pandas，则执行如下命令：

```
conda install numpy scipy pandas
```

你也可以指定安装的版本，比如安装 1.1 版本的 numpy：

```
conda install numpy=1.10
```

移除一个 package：

```
conda remove package_name
```

升级 package 版本：

```
conda update package_name
```

查看所有的 packages：

```
conda list
```

如果你记不清 package 的具体名称，也可以进行模糊查询：

```
conda search search_term
```

四、如何管理 Python 环境？

默认的环境是 root，你也可以创建一个新环境：

```
conda create -n env_name list_of_packages
```

其中 -n 代表 name，env_name 是需要创建的环境名称，list of packages 则是列出在新环境中需要安装的工具包。

例如，当我安装了 Python3 版本的 Anaconda 后，默认的 root 环境自然是 Python3，但是我还需要创建一个 Python 2 的环境来运行旧版本的 Python 代码，最好还安装了 pandas 包，于是我们运行以下命令来创建：

```
conda create -n py2 python=2.7 pandas
```

细心的你一定会发现，py2 环境中不仅安装了 pandas，还安装了 numpy 等一系列 packages，这就是使用 conda 的方便之处，它会自动为你安装相应的依赖包，而不需要你一个个手动安装。

进入名为 env_name 的环境：

```
source activate env_name
```

退出当前环境：

```
source deactivate
```

另外注意，在 Windows 系统中，使用 activate env_name 和 deactivate 来进入和退出某个环境。

删除名为 env_name 的环境：

```
conda env remove -n env_name
```

显示所有的环境：

```
conda env list
```

当分享代码的时候，同时也需要将运行环境分享给大家，执行如下命令可以将当前环境下的 package 信息存入名为 environment 的 YAML 文件中。

```
conda env export > environment.yaml
```

同样，当执行他人的代码时，也需要配置相应的环境。这时你可以用对方分享的 YAML 文件来创建一摸一样的运行环境。

```
conda env create -f environment.yaml
```

至此，你已跨入 Anaconda 的大门，后续就可以徜徉在 Python 的海洋中了。

1.2 预备知识

1.2.1 解释器

Linux/Unix的系统上，一般默认的 python 版本为 2.x，我们可以将 python3.x 安装在 **/usr/local/python3** 目录中。

安装完成后，我们可以将路径 **/usr/local/python3/bin** 添加到您的 Linux/Unix 操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python3 。

```
$ PATH=$PATH:/usr/local/python3/bin/python3      # 设置环境变量  
$ python3 --version  
Python 3.4.0
```

在Window系统下你可以通过以下命令来设置Python的环境变量，假设你的Python安装在 C:\Python34 下：

```
set path=%path%;C:\python34
```

1.2.2 编程方式

交互式编程

我们可以在命令提示符中输入"Python"命令来启动Python解释器：

```
$ python3
```

执行以上命令后，出现如下窗口信息：

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 python 提示符中输入以下语句，然后按回车键查看运行效果：

```
print ("Hello, Python!");
```

以上命令执行结果如下：

```
Hello, Python!
```

当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

```
>>> flag = True
>>> if flag :
...     print("flag 条件为 True!")
...
flag 条件为 True!
```

脚本式编程

将如下代码拷贝至 **hello.py**文件中：

```
#!/usr/bin/python3

print("Hello, World!")
```

关于实例中第一行代码**#!/usr/bin/python3** 的理解：

- 如果调用python脚本时，使用: `python script.py #!/usr/bin/python` 被忽略，等同于注释。
- 如果调用python脚本时，使用: `./script.py #!/usr/bin/python` 指定解释器的路径。
- 这句话仅仅在linux或unix系统下有作用，在windows下无论在代码里加什么都无法直接运行一个文件名后缀为.py的脚本，因为在windows下文件名对文件的打开方式起了决定性作用。
- 此外还有以下形式（推荐写法）：这种用法先在 env（环境变量）设置里查找 python 的安装路径，再调用对应路径下的解释器程序完成操作。

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

通过以下命令执行该脚本：

```
python3 hello.py
```

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：

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

然后修改脚本权限，使其有执行权限，命令如下：

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

执行以下命令：

```
./hello.py
```

1.2.3 其他

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。当然你也可以为源码文件指定不同的编码：

```
# -*- 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。

标识符

- 第一个字符必须是字母表中字母或下划线'_'。
- 标识符的其他部分由字母、数字和下划线组成。
- 标识符对大小写敏感。

在Python 3中，非-ASCII 标识符也是允许的了。

python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：

```
>>> import keyword
>>> keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', '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']
```

注释

```
#!/usr/bin/python3
# 第一个注释

# 多行注释可以用多个 ***#*** 号，还有 ***** 和 *****:
```

```
#!/usr/bin/python3
# 第一个注释
# 第二个注释

'''
第三注释
第四注释
'''

"""
第五注释
第六注释
"""

'''
```

行与缩进

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。实例如下：

```
if True:
    print ("True")
else:
    print ("False")
```

以下代码最后一行语句缩进数的空格数不一致，会导致运行错误：

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
print ("False")      # 缩进不一致，会导致运行错误
^
IndentationError: unindent does not match any outer indentation level
```

多行语句

Python 通常是一行写完一条语句，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：

```
total = item_one + \
        item_two + \
        item_three
# 在 [], {}, 或 () 中的多行语句，不需要使用反斜杠(\)，例如：

total = ['item_one', 'item_two', 'item_three',
          'item_four', 'item_five']

//或者用小括号括起来
a = (22+
      33)
```

数据类型

python中数字有四种类型：整数、布尔型、浮点数和复数。

- **int** (整数), 如 1, 只有一种整数类型 int, 表示为长整型, 没有 python2 中的 Long。
- **bool** (布尔), 如 True。
- **float** (浮点数), 如 1.23、3E-2
- **complex** (复数), 如 1 + 2j、 1.1 + 2.2j

字符串(String)

- python中单引号和双引号使用完全相同。
- 使用三引号("或""")可以指定一个**多行字符串**。
- 转义符 \'
- 反斜杠可以用来转义, 使用r可以让反斜杠不发生转义。。如 r>this is a line with \n" 则\n会显示, 并不是换行。
- 按字面意义级联字符串, 如"this " "is " "string"会被自动转换为this is string。
- 字符串可以用 + 运算符连接在一起, 用 * 运算符重复。
- Python 中的字符串有两种索引方式, 从左往右以 0 开始, 从右往左以 -1 开始。
- Python中的字符串不能改变。
- Python 没有单独的字符类型, 一个字符就是长度为 1 的字符串。
- 字符串的截取的语法格式如下: 变量[头下标:尾下标:步长]

```
word = '字符串'
sentence = "这是一个句子。"
paragraph = """这是一个段落,
可以由多行组成"""

```

```
#!/usr/bin/python3

str='Runoob'

print(str)                  # 输出字符串
print(str[0:-1])            # 输出第一个到倒数第二个的所有字符
print(str[0])                # 输出字符串第一个字符
```

```
print(str[2:5])          # 输出从第三个开始到第五个的字符
print(str[2:])           # 输出从第三个开始的后的所有字符
print(str * 2)           # 输出字符串两次
print(str + '你好')      # 连接字符串

print('-----')

print('hello\nrunoob')    # 使用反斜杠(\)+n转义特殊字符
print(r'hello\nrunoob')   # 在字符串前面添加一个 r, 表示原始字符串, 不会发生转义
```

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。

空行与代码缩进不同，空行并不是Python语法的一部分。书写时不插入空行，Python解释器运行也不会出错。但是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。

记住：空行也是程序代码的一部分。

等待用户输入

执行下面的程序在按回车键后就会等待用户输入：

```
#!/usr/bin/python3

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。一旦用户按下键时，程序将退出。

同一行显示多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

```
#!/usr/bin/python3

import sys; x = 'runoob'; sys.stdout.write(x + '\n')
```

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，我们称之为代码组。

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)。

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

Print 输出

print 默认输出是换行的，如果要实现不换行需要在变量末尾加上 **end=""**：

```
#!/usr/bin/python3  
  
x="a"  
y="b"  
# 换行输出  
print( x )  
print( y )  
  
print('-----')  
# 不换行输出  
print( x, end=" " )  
print( y, end=" " )  
print()
```

import 与 from...import

在 python 用 **import** 或者 **from...import** 来导入相应的模块。

将整个模块(somemodule)导入，格式为： **import somemodule**

从某个模块中导入某个函数,格式为： **from somemodule import somefunction**

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

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

导入 sys 模块

```
import sys  
print('=====Python import mode=====');  
print ('命令行参数为:')  
for i in sys.argv:  
    print (i)  
print ('\n python 路径为',sys.path)
```

导入 sys 模块的 argv,path 成员

```
from sys import argv, path # 导入特定的成员  
  
print('=====python from import=====')  
print('path:', path) # 因为已经导入path成员，所以此处引用时不需要加sys.path
```

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：

```
$ python -h  
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...  
Options and arguments (and corresponding environment variables):  
-c cmd : program passed in as string (terminates option list)  
-d : debug output from parser (also PYTHONDEBUG=x)  
-E : ignore environment variables (such as PYTHONPATH)  
-h : print this help message and exit  
  
[ etc. ]
```

我们在使用脚本形式执行 Python 时，可以接收命令行输入的参数，具体使用可以参照 [Python 3 命令行参数](#)。

1.3 基本数据类型

Python3 中有六个标准的数据类型：

- Number (数字)
- String (字符串)
- List (列表)
- Tuple (元组)
- Set (集合)
- Dictionary (字典)

Python3 的六个标准数据类型中：

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

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。**

等号 (=) 用来给变量赋值。

等号 (=) 运算符左边是一个变量名,等号 (=) 运算符右边是存储在变量中的值。例如：

```
#!/usr/bin/python3

counter = 100          # 整型变量
miles   = 1000.0        # 浮点型变量
name    = "runoob"      # 字符串
```

多个变量赋值

Python允许你同时为多个变量赋值。例如：

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

以上实例，创建一个整型对象，值为 1，从后向前赋值，三个变量被赋予相同的数值。

您也可以为多个对象指定多个变量。例如：

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

以上实例，两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象 "runoob" 分配给变量 c。

1.3.1 Number (数字)

定义

Python 数字数据类型用于存储数值。数据类型是不允许改变的,这就意味着如果改变数字数据类型的值，将重新分配内存空间。

以下实例在变量赋值时 Number 对象将被创建：

```
var1 = 1
var2 = 10
```

您也可以使用del语句删除一些数字对象的引用。

del语句的语法是：

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

您可以通过使用del语句删除单个或多个对象的引用，例如：

```
del var
del var_a, var_b
```

Python 支持四种不同的数值类型：

- **整型(Int)** - 通常被称为是整型或整数，是正或负整数，不带小数点。Python3 整型是没有限制大小的，可以当作 Long 类型使用，所以 Python3 没有 Python2 的 Long 类型。
- **浮点型(float)** - 浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示 ($2.5e2 = 2.5 \times 10^2 = 250$)

- **复数(complex)** - 复数由实数部分和虚数部分构成，可以用`a + bj`,或者`complex(a,b)`表示，复数的实部a和虚部b都是浮点型。
- **布尔型 bool** - Python 中布尔值使用常量 **True** 和 **False** 来表示。在数值上下文环境中，**True** 被当作 **1**，**False** 被当作 **0**，

内置的 `type()` 函数可以用来查询变量所指的对象类型。

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

我们可以使用十六进制和八进制来代表整数：

```
>>> number = 0xA0F # 十六进制
>>> number
2575

>>> number=0o37 # 八进制
>>> number
31
```

例子

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0j
-0x260	-32.54e100	3e+26j
0x69	70.2-E12	4.53e-7j

数字类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

- **int(x)** 将x转换为一个整数。
- **float(x)** 将x转换到一个浮点数。
- **complex(x)** 将x转换到一个复数，实数部分为 x，虚数部分为 0。
- **complex(x, y)** 将 x 和 y 转换到一个复数，实数部分为 x，虚数部分为 y。x 和 y 是数字表达式。

以下实例将浮点数变量 a 转换为整数：

```
>>> a = 1.0
>>> int(a)
1
```

数字运算

Python 解释器可以作为一个简单的计算器，您可以在解释器里输入一个表达式，它将输出表达式的值。

表达式的语法很直白：

- +, -, * 和 /, 和其它语言（如Pascal或C）里一样。注意：在不同的机器上浮点运算的结果可能会不一样。
- 在整数除法中，除法 / 总是返回一个浮点数，如果只想得到整数的结果，丢弃可能的分数部分，可以使用运算符 //： 但得到的并不一定是整数类型的数，它与分母分子的数据类型有关系。

```
>>> 7//2
3
>>> 7.0//2
3.0
>>> 7//2.0
3.0
>>>
```

- 等号 = 用于给变量赋值。赋值之后，除了下一个提示符，解释器不会显示任何结果。
- Python 可以使用 ** 操作来进行幂运算：

```
>>> 5 ** 2 # 5 的平方
25
```

- 变量在使用前必须先"定义"（即赋予变量一个值），否则会出现错误：

```
>>> n # 尝试访问一个未定义的变量
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

- 不同类型的数混合运算时会将整数转换为浮点数：

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

- 在交互模式中，最后被输出的表达式结果被赋值给变量 _。例如：

```

>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06

```

- 此处，`_` 变量应被用户视为只读变量。

数学函数(math)

注: 以下某些是python自带函数，并不包含在math包中，如abs, max, min...

函数	返回值(描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
cmp(x, y)	如果 $x < y$ 返回 -1, 如果 $x == y$ 返回 0, 如果 $x > y$ 返回 1。 Python 3 已废弃 。使用 使用 (x>y)-(x<y) 替换。
exp(x)	返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045
fabs(x)	返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2, ...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2, ...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x,y)	$x^{**}y$ 运算后的值。
round(x,n)	返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)	返回数字x的平方根。

随机数函数(random)

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

Python包含以下常用随机数函数：

函数	描述
choice(seq)	从序列的元素中随机挑选一个元素，比如random.choice(range(10))，从0到9中随机挑选一个整数。
randrange([start,] stop[,step])	从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为1
random()	随机生成下一个实数，它在[0,1)范围内。
[seed(x)]	改变随机数生成器的种子seed。如果你不了解其原理，你不必特别去设定seed，Python会帮你选择seed。
shuffle(lst)	将序列的所有元素随机排序
uniform(x,y)	随机生成下一个实数，它在[x,y]范围内。

三角函数(math)

函数	描述
acos(x)	返回x的反余弦弧度值。
asin(x)	返回x的反正弦弧度值。
atan(x)	返回x的反正切弧度值。
atan2(y,x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回x的弧度的余弦值。
hypot(x,y)	返回欧几里德范数 sqrt(xx + yy)。
sin(x)	返回的x弧度的正弦值。
tan(x)	返回x弧度的正切值。
degrees(x)	将弧度转换为角度,如degrees(math.pi/2)， 返回90.0
radians(x)	将角度转换为弧度

数学常量(math)

常量	描述
pi	数学常量 pi (圆周率，一般以π来表示)
e	数学常量 e，e即自然常数 (自然常数)。

1.3.2 String (字符串)

字符串是 Python 中最常用的数据类型。我们可以使用引号(' 或 ")来创建字符串。

创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'
var2 = "Runoob"
```

访问字符串中的值

Python 不支持单字符类型，单字符在 Python 中也是作为一个字符串使用。

Python 访问子字符串，可以使用方括号来截取字符串，字符串的截取的语法格式如下：

变量[头下标:尾下标]

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
	:					:

```
#!/usr/bin/python3
var1 = 'Hello World!'
var2 = "Runoob"
print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])

-----
var1[0]: H
var2[1:5]: unoo
```

字符串更新

你可以截取字符串的一部分并与其他字段拼接，如下实例：

```

#!/usr/bin/python3
var1 = 'Hello World!'
print ("已更新字符串 : ", var1[:6] + 'Runoob!')
已更新字符串 : Hello Runoob!

```

转义字符

在需要在字符串中使用特殊字符时，python用反斜杠()转义字符。如下表：

转义字符	描述
(在行尾时)	续行符
\	反斜杠符号
'	单引号
"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy代表的字符，例如：\o12代表换行
\xyy	十六进制数，yy代表的字符，例如：\x0a代表换行
\other	其它的字符以普通格式输出

字符串运算符

下表实例变量a值为字符串 "Hello"， b变量值为 "Python"：

操作符	描述	实例
+	字符串连接	a + b 输出结果： HelloPython
*	重复输出字符串	a*2 输出结果： HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 e
[:]	截取字符串中的一部分，遵循 左闭右开 原则，str[0,2] 是不包含第 3 个字符的。	a[1:4] 输出结果 ell
in	成员运算符 - 如果字符串中包含给定的字符返回 True	'H' in a 输出结果 True
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	'M' not in a 输出结果 True
r/R	原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。原始字符串除在字符串的第一个引号前加上字母 r（可以大小写）以外，与普通字符串有着几乎完全相同的语法。	print(r'\n') print(R'\n')
%	格式字符串	请看下一节内容。

字符串格式化

Python 支持格式化字符串的输出。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。

```
#!/usr/bin/python3
print ("我叫 %s 今年 %d 岁!" % ('小明', 10))
---
我叫 小明 今年 10 岁!
```

python字符串格式化符号:

符号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令:

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
	在正数前面显示空格
#	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')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

Python2.6 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。

三引号

python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。实例如下

```
#!/usr/bin/python3
para_str = """这是一个多行字符串的实例
多行字符串可以使用制表符
TAB ( \t )。
也可以使用换行符 [ \n ]。
"""
print (para_str)

---
这是一个多行字符串的实例
多行字符串可以使用制表符
TAB (     )。
也可以使用换行符 [
]。
```

三引号让程序员从引号和特殊字符串的泥潭里面解脱出来，自始至终保持一小块字符串的格式是所谓的WYSIWYG（所见即所得）格式的。

Unicode 字符串

在Python2中，普通字符串是以8位ASCII码进行存储的，而Unicode字符串则存储为16位unicode字符串，这样能够表示更多的字符集。使用的语法是在字符串前面加上前缀 **u**。

在Python3中，所有的字符串都是Unicode字符串。

字符串前加u r

python字符串前面加u, r, b的含义

<https://blog.csdn.net/u010496169/article/details/70045895>

u/U:表示unicode字符串

不是仅仅是针对中文，可以针对任何的字符串，代表是对字符串进行unicode编码。

一般英文字符在使用各种编码下，基本都可以正常解析，所以一般不带u；但是中文，必须表明所需编码，否则一旦编码转换就会出现乱码。

建议所有编码方式采用utf8

r/R:非转义的原始字符串

与普通字符相比，其他相对特殊的字符，其中可能包含转义字符，即那些，反斜杠加上对应字母，表示对应的特殊含义的，比如最常见的"\n"表示换行，"\t"表示Tab等。而如果是以r开头，那么说明后面的字符，都是普通的字符了，即如果是"\n"那么表示一个反斜杠字符，一个字母n，而不是表示换行了。

以r开头的字符，常用于正则表达式，对应着re模块。

b:bytes

python3.x里默认的str是(py2.x里的)unicode，bytes是(py2.x)的str，b""前缀代表的就是bytes
python2.x里，b前缀没什么具体意义，只是为了兼容python3.x的这种写法

参考：<http://blog.csdn.net/zhangxinrun/article/details/8124333>

http://www.oschina.net/question/437227_106832

字符串内建函数

Python 的字符串常用内建函数如下：

序号	方法及描述
1	capitalize() 将字符串的第一个字符转换为大写
2	center(width, fillchar) 返回一个指定的宽度 width 居中的字符串, fillchar 为填充的字符, 默认为空格。
3	count(str, beg=0,end=len(string)) 返回 str 在 string 里面出现的次数, 如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
4	bytes.decode(encoding="utf-8", errors="strict") Python3 中没有 decode 方法, 但我们可以使用 bytes 对象的 decode() 方法来解码给定的 bytes 对象, 这个 bytes 对象可以由 str.encode() 来编码返回。
5	encode(encoding='UTF-8',errors='strict') 以 encoding 指定的编码格式编码字符串, 如果出错默认报一个ValueError 的异常, 除非 errors 指定的是'ignore'或者'replace'
6	endswith(suffix, beg=0, end=len(string)) 检查字符串是否以 obj 结束, 如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束, 如果是, 返回 True,否则返回 False.
7	expandtabs(tabsize=8) 把字符串 string 中的 tab 符号转为空格, tab 符号默认的空格数是 8 。
8	find(str, beg=0, end=len(string)) 检测 str 是否包含在字符串中, 如果指定范围 beg 和 end , 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	index(str, beg=0, end=len(string)) 跟find()方法一样, 只不过如果str不在字符串中会报一个异常.
10	isalnum() 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 True,否则返回 False
11	isalpha() 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 True, 否则返回 False
12	isdigit() 如果字符串只包含数字则返回 True 否则返回 False..
13	islower() 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 True, 否则返回 False
14	isnumeric() 如果字符串中只包含数字字符, 则返回 True, 否则返回 False
15	isspace() 如果字符串中只包含空白, 则返回 True, 否则返回 False.
16	istitle() 如果字符串是标题化的(见 title())则返回 True, 否则返回 False
17	isupper() 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 True, 否则返回 False
18	join(seq) 以指定字符串作为分隔符, 将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
19	len(string) 返回字符串长度
20	ljust(width, fillchar) 返回一个原字符串左对齐,并使用 fillchar 填充至长度 width 的新字符串, fillchar 默认为空格。
21	lower() 转换字符串中所有大写字符为小写.
22	lstrip() 截掉字符串左边的空格或指定字符。

序号	方法及描述
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。

```
# 给字符串前面补0
n = "123"
s = n.zfill(5)
assert s == "00123"
# zfill()也可以给负数补0
n = "-123"
s = n.zfill(5)
assert s == "-0123"
```

```
# 对于纯数字，我们也可以通过格式化的方式来补0
n = 123
s = "%05d" % n
assert s == "00123"

print("{:05d}.jpg".format(12))
```

1.3.3 List (列表)

序列是Python中最基本的数据结构:

- 序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是0，第二个索引是1，依此类推。
- Python有6个序列的内置类型，但最常见的是列表和元组。
- 序列都可以进行的操作包括索引，切片，加，乘，检查成员。
- 此外，Python已经内置确定序列的长度以及确定最大和最小的元素的方法。

列表是最常用的Python数据类型，它可以作为一个方括号内的逗号分隔值出现

- 列表的数据项不需要具有相同的类型
- 创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。如下所示：

```
list1 = ['Google', 'Runoob', 1997, 2000];
list2 = [1, 2, 3, 4, 5];
list3 = ["a", "b", "c", "d"];
```

- 与字符串的索引一样，列表索引从0开始。列表可以进行截取、组合等。

访问列表中的值

使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示：

```
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])

-----
list1[0]: Google
list2[1:5]: [2, 3, 4, 5]
```

更新列表

你可以对列表的数据项进行修改或更新，你也可以使用append()方法来添加列表项，如下所示：

```
#!/usr/bin/python3
list = ['Google', 'Runoob', 1997, 2000]
print ("第三个元素为 : ", list[2])
list[2] = 2001
print ("更新后的第三个元素为 : ", list[2])
```

删除列表元素

可以使用 del 语句来删除列表的元素，如下实例

```
#!/usr/bin/python3    list = ['Google', 'Runoob', 1997, 2000]    print ("原始列表 : ", list)    del list[2]    print ("删除第三个元素 : ", list)
```

```
原始列表 :  ['Google', 'Runoob', 1997, 2000]
删除第三个元素 :  ['Google', 'Runoob', 2000]
```

注意：我们会在接下来的章节讨论 remove() 方法的使用

列表脚本操作符

列表对 + 和 * 的操作符与字符串相似。+ 号用于组合列表，* 号用于重复列表。

如下所示：

Python 表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print(x, end=" ")	1 2 3	迭代

嵌套列表

使用嵌套列表即在列表里创建其它列表，例如：

```

a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]

x
[['a', 'b', 'c'], [1, 2, 3]]

x[0]
['a', 'b', 'c']

x[0][1]
'b'

```

列表函数&方法

Python包含以下函数:

序号	函数
1	len(list) 列表元素个数
2	max(list) 返回列表元素最大值
3	min(list) 返回列表元素最小值
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	list.copy() 复制列表

1.3.4 Tuple (元组)

Python 的元组与列表类似，不同之处在于元组的元素不能修改。

- 元组使用小括号，列表使用方括号
- 元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可
- 元组与字符串类似，下标索引从0开始，可以进行截取，组合等

```
tup1 = ('Google', 'Runoob', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";    # 不需要括号也可以
type(tup3)
<class 'tuple'>

#创建空元组
tup1 = ();

#元组中只包含一个元素时，需要在元素后面添加逗号，否则括号会被当作运算符使用：
tup1 = (50)
type(tup1)      # 不加逗号，类型为整型 <class 'int'>
tup1 = (50,)
type(tup1)      # 加上逗号，类型为元组 <class 'tuple'>
```

访问元组

元组可以使用下标索引来访问元组中的值，如下实例：

```
tup1[0]: Google
tup2[1:5]: (2, 3, 4, 5)
```

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例：

```
#!/usr/bin/python3
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz')
# 以下修改元组元素操作是非法的。
# tup1[0] = 100

# 创建一个新的元组
tup3 = tup1 + tup2;
print (tup3)

---
(12, 34.56, 'abc', 'xyz')
```

删除元组

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，如下实例：

```
del tup
```

元组运算符

与字符串一样，元组之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

Python 表达式	结果	描述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
('Hi!') * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
3 in (1, 2, 3)	True	元素是否存在
for x in (1, 2, 3): print (x,)	1 2 3	迭代

元组索引，截取

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素，如下所示：

元组：

```
L = ('Google', 'Taobao', 'Runoob')
```

Python 表达式	结果	描述
L[2]	'Runoob'	读取第三个元素
L[-2]	'Taobao'	反向读取；读取倒数第二个元素
L[1:]	('Taobao', 'Runoob')	截取元素，从第二个开始后的所有元素。

元组内置函数

Python元组包含了以下内置函数

序号	方法及描述	实例
1	len(tuple) 计算元组元素个数。	>>> tuple1 = ('Google', 'Runoob', 'Taobao') >>> len(tuple1) 3 >>>
2	max(tuple) 返回元组中元素最大值。	>>> tuple2 = ('5', '4', '8') >>> max(tuple2) '8' >>>
3	min(tuple) 返回元组中元素最小值。	>>> tuple2 = ('5', '4', '8') >>> min(tuple2) '4' >>>
4	tuple(seq) 将列表转换为元组。	>>> list1= ['Google', 'Taobao', 'Runoob', 'Baidu'] >>> tuple1=tuple(list1) >>> tuple1 ('Google', 'Taobao', 'Runoob', 'Baidu')

1.3.5 Dictionary (字典)

字典是另一种可变容器模型，且可存储任意类型对象。

- 字典的每个键值(key=>value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号({})中，格式如下所示：

```
d = {key1 : value1, key2 : value2 }
```

- 键必须是唯一的，但值则不必。
- 值可以取任何数据类型，但键必须是不可变的，如字符串，数字或元组。

一个简单的字典实例：

```
dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

也可如此创建字典：

```
dict1 = { 'abc': 456 };
dict2 = { 'abc': 123, 98.6: 37 };
```

访问字典里的值

把相应的键放入到方括号中，如下实例：

```
#!/usr/bin/python3
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])

---
dict['Name']: Runoob
dict['Age']: 7
```

修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例：

```
#!/usr/bin/python3
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # 更新 Age
dict['School'] = "菜鸟教程" # 添加信息
```

删除字典元素

能删单一的元素也能清空字典，清空只需一项操作。

```
#!/usr/bin/python3
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}
del dict['Name'] # 删除键 'Name'

dict.clear()      # 清空字典
del dict         # 删除字典
```

注：`del()` 方法后面也会讨论。

字典键的特性

字典值可以是任何的 python 对象，既可以是标准的对象，也可以是用户定义的，但键不行。

- 不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例：
- 键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行

字典内置函数&方法

Python字典包含了以下内置函数：

序号	函数及描述	实例
1	len(dict) 计算字典元素个数，即键的总数。	<pre>>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> len(dict) 3</pre>
2	str(dict) 输出字典，以可打印的字符串表示。	<pre>>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> str(dict) "{'Name': 'Runoob', 'Class': 'First', 'Age': 7}"</pre>
3	type(variable) 返回输入的变量类型，如果变量是字典就返回字典类型。	<pre>>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> type(dict) <class 'dict'></pre>

Python字典包含了以下内置方法：

序号	函数及描述
1	radiansdict.clear() 删除字典内所有元素
2	radiansdict.copy() 返回一个字典的浅复制
3	radiansdict.fromkeys() 创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
4	radiansdict.get(key, default=None) 返回指定键的值，如果值不在字典中返回default值
5	key in dict 如果键在字典dict里返回true，否则返回false
6	radiansdict.items() 以列表返回可遍历的(键, 值) 元组数组
7	radiansdict.keys() 返回一个迭代器，可以使用 list() 来转换为列表
8	radiansdict.setdefault(key, default=None) 和get()类似, 但如果键不存在于字典中, 将会添加键并将值设为default
9	radiansdict.update(dict2) 把字典dict2的键/值对更新到dict里
10	radiansdict.values() 返回一个迭代器，可以使用 list() 来转换为列表
11	[pop(key, default)] 删除字典给定键 key 所对应的值，返回值为被删除的值。key值必须给出。否则，返回default值。
12	popitem() 随机返回并删除字典中的一对键和值(一般删除末尾对)。

字典元素排序

```
sorted(dict.keys())
for key in sorted(dict.keys())
```

<https://blog.csdn.net/tangtanghao511/article/details/47810729>

<https://blog.csdn.net/lilongsy/article/details/72545638>

<https://blog.csdn.net/buster2014/article/details/50939892>

1.3.6 Set (集合)

集合 (set) 是一个无序的不重复元素序列。

- 可以使用大括号 {} 或者 set() 函数创建集合
- 创建一个空集合必须用 set() 而不是 {}, 因为 {} 是用来创建一个空字典。

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)                                     # 这里演示的是去重功能
{'orange', 'banana', 'pear', 'apple'}

'orange' in basket                               # 快速判断元素是否在集合内
True
'crabgrass' in basket
False

# 下面展示两个集合间的运算.
a = set('abracadabra')
b = set('alacazam')
a
{'a', 'r', 'b', 'c', 'd'}
a - b                                         # 集合a中包含而集合b中不包含的元素
{'r', 'd', 'b'}
a | b                                         # 集合a或b中包含的所有元素
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
a & b                                         # 集合a和b中都包含了的元素
{'a', 'c'}
a ^ b                                         # 不同时包含于a和b的元素
{'r', 'd', 'b', 'm', 'z', 'l'}
```

- 类似列表推导式，同样集合支持集合推导式(Set comprehension):

```
a = {x for x in 'abracadabra' if x not in 'abc'} >>> a {'r', 'd'}
```

添加元素

```
s.add( x )
```

将元素 x 添加到集合 s 中，如果元素已存在，则不进行任何操作。

- 还有一个方法，也可以添加元素，且参数可以是列表，元组，字典等，语法格式如下：

```
s.update( x )
```

x 可以有多个，用逗号分开。

```
thisset = set(("Google", "Runoob", "Taobao")) # set()只支持一个参数
thisset.update({1,3})
print(thisset)
{1, 3, 'Google', 'Taobao', 'Runoob'}
thisset.update([1,4],[5,6])
print(thisset)
{1, 3, 4, 5, 6, 'Google', 'Taobao', 'Runoob'}
```

移除元素

```
s.remove( x )
```

将元素 x 从集合 s 中移除，如果元素不存在，则会发生错误。

```
s.discard( x )
```

我们也可以设置随机删除集合中的一个元素，语法格式如下：

```
s.pop()
```

然而在交互模式，pop 是删除集合的第一个元素（排序后的集合的第一个元素）。

计算集合元素个数

```
len(s)
```

计算集合 s 元素个数。

清空集合

```
s.clear()
```

清空集合 s。

判断元素是否在集合中存在

```
x in s
```

判断元素 x 是否在集合 s 中，存在返回 True，不存在返回 False。

集合内置方法完整列表

方法	描述
add()	为集合添加元素
clear()	移除集合中的所有元素
copy()	拷贝一个集合
difference()	返回多个集合的差集
difference_update()	移除集合中的元素，该元素在指定的集合也存在。
discard()	删除集合中指定的元素
intersection()	返回集合的交集
intersection_update()	删除集合中的元素，该元素在指定的集合中不存在。
isdisjoint()	判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。
issubset()	判断指定集合是否为该方法参数集合的子集。
issuperset()	判断该方法的参数集合是否为指定集合的子集
pop()	随机移除元素
remove()	移除指定元素
symmetric_difference()	返回两个集合中不重复的元素集合。
symmetric_difference_update()	移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。
union()	返回两个集合的并集
update()	给集合添加元素

1.3.7 数据类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

函数	描述
<code>[int(x ,base])</code>	将x转换为一个整数
<code>float(x)</code>	将x转换到一个浮点数
<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>	转换为可变集合
<code>dict(d)</code>	创建一个字典。d 必须是一个序列 (key,value)元组。
<code>frozenset(s)</code>	转换为不可变集合
<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

1.4 运算符

算术运算符

以下假设变量a为10， 变量b为21：

运算符	描述	实例
<code>+</code>	加 - 两个对象相加	<code>a + b</code> 输出结果 31
<code>-</code>	减 - 得到负数或是一个数减去另一个数	<code>a - b</code> 输出结果 -11
<code>*</code>	乘 - 两个数相乘或是返回一个被重复若干次的字符串	<code>a * b</code> 输出结果 210
<code>/</code>	除 - x 除以 y	<code>b / a</code> 输出结果 2.1
<code>%</code>	取模 - 返回除法的余数	<code>b % a</code> 输出结果 1
<code>**</code>	幂 - 返回x的y次幂	<code>a**b</code> 为10的21次方
<code>//</code>	取整除 - 返回商的整数部分	<code>9//2</code> 输出结果 4 , <code>9.0//2.0</code> 输出结果 4.

比较运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 True。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。	(a < b) 返回 True。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 True。

赋值运算符

以下假设变量a为10，变量b为20：

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

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：

下表中变量 a 为 60, b 为 13二进制格式如下：

```
> a = 0011 1100  
  
> b = 0000 1101  
  
> \-----  
  
> a&b = 0000 1100  
  
> a|b = 0011 1101  
  
> a\^b = 0011 0001  
  
> ~a = 1100 0011
```

运算符	描述	实例
&	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	(a & b) 输出结果 12 , 二进制解释： 0000 1100
	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a b) 输出结果 61 , 二进制解释： 0011 1101
\^	按位异或运算符：当两对应的二进位相异时，结果为1	(a \^ b) 输出结果 49 , 二进制解释： 0011 0001
~	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。~x 类似于 -x-1	(~a) 输出结果 -61 , 二进制解释： 1100 0011, 在一个有符号二进制数的补码形式。
<<	左移动运算符：运算数的各二进位全部左移若干位，由" <<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 , 二进制解释： 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位， ">>"右边的数指定移动的位数	a >> 2 输出结果 15 , 二进制解释： 0000 1111

逻辑运算符

Python语言支持逻辑运算符，以下假设变量 a 为 10, b 为 20:

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False, x and y 返回 false(0), 为真它返回 y 的计算值。	(a and b) 返回 20。
or	x or y	布尔"或" - 如果 x 是 True, 它返回 x 的值, 为假它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True, 返回 False。如果 x 为 False, 它返回 True。	not(a and b) 返回 False

成员运算符

除了以上的一些运算符之外，Python还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回 True, 否则返回 False。	x 在 y 序列中, 如果 x 在 y 序列中返回 True。
not in	如果在指定的序列中没有找到值返回 True, 否则返回 False。	x 不在 y 序列中, 如果 x 不在 y 序列中返回 True。

以下实例演示了Python所有成员运算符的操作：

身份运算符

身份运算符用于比较两个对象的存储单元(python 没有定义变量类型的概念，赋给了某个变量一个值，便开辟了一块内存空间用于存储此变量，变量随之被定义)

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y , 类似 id(x) == id(y) , 如果引用的是同一个对象则返回 True, 否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y , 类似 id(a) != id(b) 。如果引用的不是同一个对象则返回结果 True, 否则返回 False。

注： [id\(\)](#) 函数用于获取对象内存地址。

以下实例演示了Python所有身份运算符的操作：

```
#!/usr/bin/python3

a = 20
b = 20

if ( a is b ):
    print ("1 - a 和 b 有相同的标识")
else:
    print ("1 - a 和 b 没有相同的标识")

if ( id(a) == id(b) ):
    print ("2 - a 和 b 有相同的标识")
else:
    print ("2 - a 和 b 没有相同的标识")

# 修改变量 b 的值
b = 30
if ( a is b ):
    print ("3 - a 和 b 有相同的标识")
else:
    print ("3 - a 和 b 没有相同的标识")

if ( a is not b ):
    print ("4 - a 和 b 没有相同的标识")
else:
    print ("4 - a 和 b 有相同的标识")
以上实例输出结果：

1 - a 和 b 有相同的标识
2 - a 和 b 有相同的标识
3 - a 和 b 没有相同的标识
4 - a 和 b 没有相同的标识
```

运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +\@ 和 -\@)
* / % //	乘, 除, 取模和取整除
+ -	加法减法
>> <<	右移, 左移运算符
&	位 'AND'
\^	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //=-+= *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

1.5 基本语句

1.5.1 if 条件语句

if语句的一般形式如下所示：

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

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。

```
#!/usr/bin/python3
```

```

age = int(input("请输入你家狗狗的年龄: "))
print("")
if age < 0:
    print("你是在逗我吧!")
elif age == 1:
    print("相当于 14 岁的人。")
elif age == 2:
    print("相当于 22 岁的人。")
elif age > 2:
    human = 22 + (age - 2)*5
    print("对应人类年龄: ", human)

### 退出提示
input("点击 enter 键退出")

```

以下为if中常用的操作运算符:

操作符	描述
<	小于
<=	小于或等于
>	大于
>=	大于或等于
==	等于， 比较两个值是否相等
!=	不等于

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。

```

if 表达式1:
    语句
    if 表达式2:
        语句
    elif 表达式3:
        语句
    else:
        语句
elif 表达式4:
    语句
else:
    语句

```

1.5.2 While 循环语句

Python中while语句的一般形式：

```
while 判断条件:
```

```
    语句
```

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

无限循环

我们可以通过设置条件表达式永远不为 false 来实现无限循环，实例如下：

```
#!/usr/bin/python3
var = 1
while var == 1 : # 表达式永远为 true
    num = int(input("输入一个数字 :"))
    print ("你输入的数字是: ", num)

print ("Good bye!")
```

你可以使用 **CTRL+C** 来退出当前的无限循环。

无限循环在服务器上客户端的实时请求非常有用。

while ... else 语句

在 while ... else 在条件语句为 false 时执行 else 的语句块：

```
#!/usr/bin/python3
count = 0
while count < 5:
    print (count, " 小于 5")
    count = count + 1
else:
    print (count, " 大于或等于 5")

---
0 小于 5
1 小于 5
2 小于 5
3 小于 5
4 小于 5
5 大于或等于 5
```

1.5.3 for 语句

Python 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：

```
for <variable> in <sequence>:  
    <statements>  
else:  
    <statements>
```

以下 for 实例中使用了 break 语句，break 语句用于跳出**当前循环体**：

```
#!/usr/bin/python3  
  
sites = ["Baidu", "Google", "Runoob", "Taobao"]  
for site in sites:  
    if site == "Runoob":  
        print("菜鸟教程!")  
        break  
    print("循环数据 " + site)  
else:  
    print("没有循环数据!")  
print("完成循环!")
```

range()函数

如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如：

```
for i in range(5):  
    print(i)  
  
for i in range(5, 9) :  
    print(i)  
  
for i in range(0, 10, 3) :  
    print(i)  
  
for i in range(-10, -100, -30) :  
    print(i)  
  
a = ['Google', 'Baidu', 'Runoob', 'Taobao', 'QQ']  
for i in range(len(a)):  
    print(i, a[i])  
  
# 还可以使用range()函数来创建一个列表：  
list(range(5))  
[0, 1, 2, 3, 4]
```

1.5.4 break和continue语句及循环中的else子句

- break 语句可以跳出 for 和 while 的循环体。如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。
- continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

实例

```
#!/usr/bin/python3

for letter in 'Runoob':      # 第一个实例
    if letter == 'b':
        break
    print ('当前字母为 :', letter)

var = 10                      # 第二个实例
while var > 0:
    var = var -1
    if var == 5:              # 变量为 5 时跳过输出
        continue
    print ('当前变量值 :', var)
print ("Good bye!")

for n in range(2, 10):  # 查询质数
    for x in range(2, n):
        if n % x == 0:
            print(n, '等于', x, '*', n//x)
            break
    else:
        # 循环中没有找到元素
        print(n, '是质数')
```

1.5.5 pass 语句

Python pass是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句，如下实例

```
while True:
...     pass  # 等待键盘中断 (Ctrl+C)

#最小的类:
class MyEmptyClass:
...     pass

# 以下实例在字母为 o 时 执行 pass 语句块:
for letter in 'Runoob':
    if letter == 'o':
        pass           # 注意: 执行pass并不是后面的语句就不执行, 与continue区分!
        print ('执行 pass 块')
    print ('当前字母 :', letter)

print ("Good bye!")
```

1.5.6 assert 语句

assert 语句用来判定一个表达式是否为真，不是的话则会抛出异常。常用于if语句之前，用来提示用户出现了if语句不处理的情况。

```
# assert的语法格式:  
assert expression  
  
# 它的等价语句为:  
if not expression:  
    raise AssertionError  
  
# 这段代码用来检测数据类型的断言，因为 a_str 是 str 类型，所以认为它是 int 类型肯定会引发错误。  
>>> a_str = 'this is a string'  
>>> type(a_str)  
<type 'str'>  
>>> assert type(a_str)== str  
>>> assert type(a_str)== int  
  
Traceback (most recent call last):  
  File "<pyshell#41>", line 1, in <module>  
    assert type(a_str)== int  
AssertionError
```

1.6 迭代器与生成器

1.6.1 迭代器

定义

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()**
- 字符串，列表或元组对象都可用于创建迭代器：

```
>>>list=[1,2,3,4]  
>>> it = iter(list)      # 创建迭代器对象  
>>> print (next(it))   # 输出迭代器的下一个元素  
1  
>>> print (next(it))  
2  
# 迭代到最后一个元素后会抛出异常 StopIteration  
  
# 迭代器对象可以使用常规for语句进行遍历：  
# 其实序列对象都是可以直接遍历的  
for x in it:  
    print (x, end=" ")
```

```
# 也可以使用next() 函数
import sys          # 引入 sys 模块

while True:
    try:
        print(next(it))
    except StopIteration:
        sys.exit()
```

创建一个迭代器

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__()`。

- 如果你已经了解的面向对象编程，就知道类都有一个构造函数，Python 的构造函数为 `__init__()`，它会在对象初始化的时候执行。更多内容查阅：[Python3 面向对象](#)
- `__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（Python 2 里是 `next()`）会返回下一个迭代器对象。

```
# 创建一个返回数字的迭代器，初始值为 1，逐步递增 1:
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

`StopIteration` 异常用于标识迭代的完成，防止出现无限循环的情况，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。

在 20 次迭代后停止执行：

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
```

```

    return self

def __next__(self):
    if self.a <= 20:
        x = self.a
        self.a += 1
        return x
    else:
        raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
    print(x)

```

1.6.2 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。

以下实例使用 `yield` 实现斐波那契数列：

```

#!/usr/bin/python3

import sys

def fibonacci(n): # 生成器函数 - 斐波那契
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(10) # f 是一个迭代器，由生成器返回生成

while True: # 不断使用next()函数返回迭代器的值
    try:
        print (next(f), end=" ")
    except StopIteration:
        sys.exit()

```

1.7 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。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

1.7.1 定义函数

你可以定义一个由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，并且缩进。
- **return [表达式]** 结束函数，选择性地返回一个值给调用方。不带表达式的return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：

```
def 函数名 (参数列表) :  
    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。

```
#!/usr/bin/python3  
  
# 计算面积函数  
def area(width, height):  
    return width * height  
  
def print_welcome(name):  
    print("Welcome", name)  
  
print_welcome("Runoob")  
w = 4  
h = 5  
print("width =", w, " height =", h, " area =", area(w, h))
```

1.7.2 函数调用

定义一个函数：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，你可以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。

```
#!/usr/bin/python3

# 定义函数
def printme( str ):
    # 打印任何传入的字符串
    print (str)
    return

# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

1.7.3 参数传递

在 python 中，类型属于对象，变量是没有类型的：

```
a=[1,2,3]
a="Runoob"
```

以上代码中，**[1,2,3]** 是 List 类型，"Runoob" 是 String 类型，而变量 a 是没有类型，她仅仅是一个对象的引用（一个指针），可以是指向 List 类型对象，也可以是指向 String 类型对象。

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- **不可变类型：**变量赋值 **a=5** 后再赋值 **a=10**，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变a的值，相当于新生成了a。
- **可变类型：**变量赋值 **la=[1,2,3,4]** 后再赋值 **la[2]=5** 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。
 - la这个指针并没有改变，同样，再将la赋值给另一个变量lb时，只是指针的拷贝，并没有新对象生成，所以二者指向了同一个对象的地址

```
list=[1, 2, 3, 4]
b=list
list[2]=0
>>> list
[1, 2, 0, 4]
>>> b
[1, 2, 0, 4]
```

- 但不可变类型，赋值便是新生成了对象，无论怎么对变量(指针)互相拷贝，二者之间没有关系，因为已经改变了，虽然都是叫a, 但地址值一旦改变之前的a便被丢弃了

```
>>> a=1 # a指针指向一个对象
>>> b=a # b拷贝a的地址，指向同一个对象1
>>> a=2 # 丢弃旧指针a，生成新的指针，指向第二个对象
>>> b    # b仍指向旧的地址
1
```

python 函数的参数传递：

- **不可变类型**：类似 c++ 的值传递，如 整数、字符串、元组。如 fun (a) ， 传递的只是a的值，没有影响a对象本身。比如在 fun (a) 内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。
- **可变类型**：类似 c++ 的引用传递，如 列表，字典。如 fun (la) ， 则是将 la 真正的传过去，修改后fun外部的la 也会受影响
- python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

传不可变对象实例

```
#!/usr/bin/python3

def ChangeInt( a ):
    a = 10

b = 2
ChangeInt(b)
print( b ) # 结果是 2
```

实例中有 int 对象 2，指向它的变量是 b，在传递给 ChangeInt 函数时，按传值的方式复制了变量 b，a 和 b 都指向了同一个 Int 对象，在 a=10 时，则新生成一个 int 值对象 10，并让 a 指向它。

传可变对象实例

可变对象在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。(其实根本区别还是对象类型是可变的还是不可变的，不可变的会创建新对象，而可变的不会创建新对象)

```
#!/usr/bin/python3

# 可写函数说明
def changeme( mylist ):
    "修改传入的列表"
    mylist.append([1,2,3,4])
    print ("函数内取值: ", mylist)
    return

# 调用changeme函数
mylist = [10,20,30]
changeme( mylist )
print ("函数外取值: ", mylist)

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

1.7.4 参数

以下是调用函数时可使用的正式参数类型：

- 必需参数
- 关键字参数
- 默认参数
- 不定长参数

必需参数

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。

调用printme()函数，你必须传入一个参数，不然会出现语法错误：

```
#!/usr/bin/python3

#可写函数说明
def printme( str ):
    "打印任何传入的字符串"
    print (str)
    return

#调用printme函数
printme()
```

关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

以下实例在函数 printme() 调用时使用参数名：

```
printme( str = "菜鸟教程" )
```

默认参数

调用函数时，如果没有传递参数，则会使用默认参数。以下实例中如果没有传入 age 参数，则使用默认值：

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
    "函数_文档字符串"
    function_suite
    return [expression]
```

- 加了星号 * 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。

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, *vartuple ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    print (vartuple)

# 调用printinfo 函数
printinfo( 70, 60, 50 )

---
输出:
70
(60, 50)
```

- 如果在函数调用时没有指定参数，它就是一个空元组。我们也可以不向函数传递未命名的变量。
- 加了两个星号 ** 的参数会以字典的形式导入。

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, **vardict ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    print (vardict)

# 调用printinfo 函数
printinfo(1, a=2, b=3)

---
输出:
1
{'a': 2, 'b': 3}
```

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

1.7.5 匿名函数

python 使用 lambda 来创建匿名函数。

所谓匿名，意即不再使用 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。
- 虽然lambda函数看起来只能写一行，却不等同于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，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

```
#!/usr/bin/python3

# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2

# 调用sum函数
print ("相加后的值为 : ", sum( 10, 20 ))
print ("相加后的值为 : ", sum( 20, 20 ))
```

1.7.6 return 语句

return [表达式] 语句用于退出函数，选择性地向调用方返回一个表达式。不带参数值的return语句返回None。之前的例子都没有示范如何返回数值，以下实例演示了 return 语句的用法：

```
#!/usr/bin/python3

# 可写函数说明
def sum( arg1, arg2 ):
    # 返回2个参数的和."
    total = arg1 + arg2
    print ("函数内 : ", total)
    return total

# 调用sum函数
total = sum( 10, 20 )
print ("函数外 : ", total)
```

1.7.7 变量作用域

Python 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。Python的作用域一共有4种，分别是：

- L (Local) 局部作用域
- E (Enclosing) 闭包函数外的函数中
- G (Global) 全局作用域
- B (Built-in) 内置作用域 (内置函数所在模块的范围)

以 L -> E -> G -> B 的规则查找，即：在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内置中找。

```
g_count = 0 # 全局作用域
def outer():
    o_count = 1 # 闭包函数外的函数中
    def inner():
        i_count = 2 # 局部作用域
```

内置作用域是通过一个名为 `builtin` 的标准模块来实现的，但是这个变量名自身并没有放入内置作用域内，所以必须导入这个文件才能够使用它。在Python3.0中，可以使用以下的代码来查看到底预定义了哪些变量：

```
>>> import builtins
>>> dir(builtins)
```

Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 `if/elif/else/`、`try/except`、`for/while` 等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码：

```
>>> if True:
...     msg = 'I am from Runoob'
...
>>> msg
'I am from Runoob'
>>>
```

实例中 `msg` 变量定义在 `if` 语句块中，但外部还是可以访问的。

如果将 `msg` 定义在函数中，则它就是局部变量，外部不能访问：

```
>>> def test():
...     msg_inner = 'I am from Runoob'
...
>>> msg_inner
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'msg_inner' is not defined
>>>
```

从报错的信息上看，说明了 `msg_inner` 未定义，无法使用，因为它是局部变量，只有在函数内可以使用。

全局变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。

global 和 nonlocal关键字

- 当内部作用域想修改外部作用域的变量时，就要用到`global`和`nonlocal`关键字了。

以下实例修改全局变量 num：

```
#!/usr/bin/python3

num = 1
def fun1():
    global num  # 需要使用 global 关键字声明
    print(num)
    num = 123
    print(num)
fun1()
print(num)

---
```

以上实例输出结果：

```
1
123
123
```

- 如果要修改嵌套作用域（enclosing 作用域，外层非全局作用域）中的变量则需要 nonlocal 关键字了，如下实例：

```
#!/usr/bin/python3

def outer():
    num = 10
    def inner():
        nonlocal num  # nonlocal关键字声明
        num = 100
        print(num)
    inner()
    print(num)
outer()

---
```

```
100
100
```

1.8 数据结构

1.8.1 列表

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

方法	描述
list.append(x)	把一个元素添加到列表的结尾，相当于 <code>a[len(a):] = [x]</code> 。
list.extend(L)	通过添加指定列表的所有元素来扩充列表，相当于 <code>a[len(a):] = L</code> 。
list.insert(i, x)	在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 <code>a.insert(0, x)</code> 会插入到整个列表之前，而 <code>a.insert(len(a), x)</code> 相当于 <code>a.append(x)</code> 。
list.remove(x)	删除列表中值为 <code>x</code> 的第一个元素。如果没有这样的元素，就会返回一个错误。
list.pop([i])	从列表的指定位置移除元素，并将其返回。如果没有指定索引， <code>a.pop()</code> 返回最后一个元素。元素随即从列表中被移除。（方法中 <code>i</code> 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）
list.clear()	移除列表中的所有项，等于 <code>del a[:]</code> 。
list.index(x)	返回列表中第一个值为 <code>x</code> 的元素的索引。如果没有匹配的元素就会返回一个错误。
list.count(x)	返回 <code>x</code> 在列表中出现的次数。
list.sort()	对列表中的元素进行排序。
list.reverse()	倒排列表中的元素。
list.copy()	返回列表的浅复制，等于 <code>a[:]</code> 。

下面示例演示了列表的大部分方法：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

将列表当堆栈使用

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：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

将列表当队列使用

也可以把列表当做队列用，只是在队列里第一加入的元素，第一个取出来；但是拿列表用作这样的目的效率不高。在列表的最后添加或者弹出元素速度快，然而在列表里插入或者从头部弹出速度却很慢（因为所有其他的元素都得一个一个地移动）。

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()              # The first to arrive now leaves
'Eric'
>>> queue.popleft()              # The second to arrive now leaves
'John'
>>> queue                      # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

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。如果希望表达式推导出一个元组，就必须使用括号。

```
# 这里我们将列表中每个数值乘三，获得一个新的列表：
>>> vec = [2, 4, 6]
```

```

>>> [3*x for x in vec]
[6, 12, 18]

#现在我们玩一点小花样:
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]

#这里我们对序列里每一个元素逐个调用某方法:
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']

# 我们可以用 if 子句作为过滤器:
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]

#以下是一些关于循环和其它技巧的演示:
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]

# 列表推导式可以使用复杂表达式或嵌套函数:
>>> [str(round(355/113, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']

```

嵌套列表解析

```

#Python的列表还可以嵌套。

#以下实例展示了3X4的矩阵列表:
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]

#以下实例将3X4的矩阵列表转换为4X3列表:
# 因为内部[]的存在, 会先执行for i in range(4), 因为他是一个语句, 从外往里看
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

#以下实例也可以使用以下方法来实现:
>>> transposed = []

```

```
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

# 另外一种实现方法:
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

del 语句

使用 del 语句可以从一个列表中依索引而不是值来删除一个元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]

# 也可以用 del 删除实体变量:
>>> del a
```

1.8.2 元组和序列

#元组由若干逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
#如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号，不过括号通常是非常必要的（如果元组是更大的表达式的一部分）。
```

1.8.3 集合

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {}；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # 删除重复的
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                # 检测成员
True
>>> 'crabgrass' in basket
False

>>> # 以下演示了两个集合的操作
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                # a 中唯一的字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                            # 在 a 中的字母，但不在 b 中
{'r', 'd', 'b'}
>>> a | b                            # 在 a 或 b 中的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                            # 在 a 和 b 中都有的字母
{'a', 'c'}
>>> a ^ b                            # 在 a 或 b 中的字母，但不同时在 a 和 b 中
{'r', 'd', 'b', 'm', 'z', 'l'}
集合也支持推导式：
```

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

1.8.4 字典

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False

# 构造函数 dict() 直接从键值对 元组列表 中构建字典。如果有固定的模式，列表推导式指定特定的键值对：
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}

# 此外，字典推导可以用来创建任意键和值的表达式词典：
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

#如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

遍历技巧

```

#在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)

#在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)

#同时遍历两个或更多的序列，可以使用 zip() 组合：
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
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：
>>> for i in reversed(range(1, 10, 2)):
...     print(i)

#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：
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)

```

1.8.5 数据拷贝

1. 使用不可变数据类型自带的方法，如list.copy() python3版本才有
2. Python中的对象之间赋值时是按引用传递的，如果需要拷贝对象，需要使用标准库中的copy模块。
 - 1、copy.copy 浅拷贝 只拷贝父对象，不会拷贝对象的内部的子对象。
 - 2、copy.deepcopy 深拷贝 拷贝对象及其子对象

```

>>> import copy
>>> a = [1,2,3,4,['a','b']] #原始对象

>>> b = a #赋值，传对象的引用

>>> c = copy.copy(a)

>>> d = copy.deepcopy(a)

>>> a.append(5)
>>> a[4].append('c')

>>> print 'a=',a
a= [1, 2, 3, 4, ['a', 'b', 'c'], 5]
>>> print 'b=',b
b= [1, 2, 3, 4, ['a', 'b', 'c'], 5]

```

```
>>> print 'c=',c
c= [1, 2, 3, 4, ['a', 'b', 'c']]
>>> print 'd=',d
d= [1, 2, 3, 4, ['a', 'b']]
```

1.9 模块

1.9.1 模块基础

在前面的几个章节中我们脚本上是用 python 解释器来编程，如果你从 Python 解释器退出再进入，那么你定义的所有方法和变量就都消失了。为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

- 模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 python 标准库的方法。
- 使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，但是也要注意，尽量不要与内置函数名字冲突。
- 进一步如果模块名相同，我们定义不同的包（package）来分别导入不同的模块。

使用 python 标准库中模块的例子：

```
#!/usr/bin/python3
# 文件名: using_sys.py

import sys

print('命令行参数如下：')
for i in sys.argv:
    print(i)

print('\n\nPython 路径为：', sys.path, '\n')

# 命令行执行 python using_sys.py 参数1 参数2
# ===== output =====
命令行参数如下：
using_sys.py
参数1
参数2

Python 路径为： ['', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-linux-gnu', '/usr/lib/python3.5/lib-dynload',
'/home/ubuntu16/.local/lib/python3.5/site-packages', '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages']
```

- import sys 引入 python 标准库中的 sys.py 模块；这是引入某一模块的方法。
- sys.argv 是一个包含命令行参数的列表，列表第一个元素为当前文件名
- sys.path 包含了一个 Python 解释器自动查找所需模块的路径的列表，列表第一个元素为当前文件所在路径。当使用import 导入模块时，系统会在自动按照此列表中提供的路径的顺序进行搜索相应模块。
- 注意，只有使用python解释器执行某个文件时才会由sys.argv 和sys.path 变量，和被执行的文件相关，使用 import 导入，则不会产生此变量

import 语句

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[, ... moduleN]]
```

- 一个模块只会被导入一次，不管你执行了多少次import。这样可以防止导入模块被一遍又一遍地执行。
- 当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。
 - 搜索路径是一个解释器会先进行搜索的所有目录的列表，是由一系列目录名组成的，Python解释器就依次从这些目录中去寻找所引入的模块
 - 这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。
 - 搜索路径是在Python编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在sys模块中的path变量，了解了搜索路径的概念，就可以在脚本中修改sys.path来引入一些不在搜索路径中的模块

```
import sys,os  
sys.path.append(BASE_DIR)
```

from ... import 语句

Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中，语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，要导入模块 fibo 的 fib 函数，使用如下语句：

```
>>> from fibo import fib, fib2  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这个声明不会把整个fibo模块导入到当前的命名空间中，它只会将fibo里的fib函数引入进来。

from ... import * 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。大多数情况，Python程序员不使用这种方法，因为引入的其它来源的命名，很可能覆盖了已有的定义。

1.9.2 深入模块

__name__ 属性

- 每个模块都有一个 `__name__` 属性，如果我们直接执行.py文件，其值是 '`__main__`' 如果我们从另一个.py文件使用import将其导入，其值是.py的文件名。
- 如果我们想在模块被引入时，模块中的某一程序块不执行，比如该模块的调试代码，可以将调试代码写在if语句块内，使得该模块仅在直接执行时运行。

```
#!/usr/bin/python3
# Filename: using_name.py

if __name__ == '__main__':
    print('程序自身在运行')
else:
    print('我来自另一模块')
```

运行输出如下：

```
$ python using_name.py
程序自身在运行
$ python
>>> import using_name
我来自另一模块
>>>
```

dir() 函数

内置的函数 `dir()` 可以找到模块内定义的所有名称。以一个字符串列表的形式返回：

```
</p>
<pre>
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '__clear_type_cache__', '__current_frames__', '__debugmallocstats__', '__getframe__',
 '__home__', '__mercurial__', '__xoptions__', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'getttotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setsswitchinterval', 'setattr', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

如果没有给定参数，那么 `dir()` 函数会罗列出当前定义的所有名称：

```

>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir() # 得到一个当前模块中定义的属性列表
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
>>> a = 5 # 建立一个新的变量 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # 删除变量名a
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>>

```

包

包是一种管理 Python 模块命名空间的形式，采用".模块名称"。

- 比如一个模块的名称是 A.B，那么他表示一个包 A中的子模块 B。就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，**采用点模块名称这种形式也不用担心不同库之间的模块重名的情况**。这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。
- 在导入一个包的时候，**Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录**。
 - 目录只有包含一个叫做 `__init__.py` 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。
 - 最简单的情况，放一个空的 :file:`__init__.py`就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的） `__all__` 变量赋值。
- 不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。
 - 现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如：.wav, :file:.aiff, :file:.au, ），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。
 - 并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）：

sound/	顶层包
__init__.py	初始化 sound 包
formats/	文件格式转换子包
__init__.py	
wav.py	
...	
effects/	声音效果子包
__init__.py	
echo.py	
test.py	
...	
test/	测试子包
__init__.py	
test.py	

```
demo.py  
main.py
```

```
# 情况1：导入当前目录及其子目录中的模块，支持多级目录  
>>> import sound.demo  
>>> import sound.formats.wav  
>>> from sound.effects import test  
>>> from sound.test import test # 相同的模块名被不同的包分隔开分别导入  
# 注意：如果使用形如import item.subitem.subsubitem这种导入形式，除了最后一项，都必须是包，而最后一项则可以是模块或者是包，但是不可以是类，函数或者变量的名字。  
  
# 情况2：从子目录中导入上级目录中的模块（如formats 目录下）  
# 首先添加上级目录到path中  
import sys  
sys.path.append("../")  
# 也可以加入绝对目录，这样系统就可以索引这个目录下的所有模块  
import effects.echo  
  
# 情况3： effects文件夹下的test文件与上一级的包test重名，如何导入test包中的test模块？  
  
##### 待解决  
如何导入与系统模块重名的模块  
  
from __future__ import absolute_import 到底是什么作用？
```

获取目录

```
import os  
  
print '***获取当前目录***'  
print os.getcwd()  
print os.path.abspath(os.path.dirname(__file__))  
  
print '***获取上级目录***'  
print os.path.abspath(os.path.dirname(os.path.dirname(__file__)))  
print os.path.abspath(os.path.dirname(os.getcwd()))  
print os.path.abspath(os.path.join(os.getcwd(), ".."))  
  
print '***获取上上级目录***'  
print os.path.abspath(os.path.join(os.getcwd(), "../.."))
```

从一个包中导入*

设想一下，如果我们使用 `from sound.effects import *`会发生什么？

- Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。但是很不幸，这个方法在 Windows平台上工作的就不是非常好，因为Windows是一个大小写不区分的系统。在这类平台上，没有人敢担保一个叫做 ECHO.py 的文件导入为模块 echo 还是 Echo 甚至 ECHO。 （例如，Windows 95就很讨厌的把每一个文件的首字母大写显示）而且 DOS 的 8+3 命名规则对长模块名称的处理会把问题搞得更纠结。

- 导入语句遵循如下规则：如果包定义文件 `__init__.py` 存在一个叫做 `__all__` 的列表变量，那么在使用 `from package import *` 的时候就把这个列表中的所有名字作为包内容导入。
- 作为包的作者，可别忘了在更新包之后保证 `all` 也更新了啊。你说我就不这么做，我就不使用导入*这种用法，好吧，没问题，谁让你是老板呢。这里有一个例子，在: `file:sounds/effects/__init__.py` 中包含如下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用`from sound.effects import *`这种用法时，你只会导入包里面这三个子模块。

常用模块

开源模块

1. <https://pypi.python.org/pypi>

future 模块

作用：

- 因python2 与 python3并不兼容，为了保证其兼容性，即在python2.1之前的版本中也可以使用python3中的一些语言特性，可以导入future模块。

```
from __future__ import *
```

常用：

```
from __future__ import print_function
from __future__ import division
from __future__ import absolute_import
```

1. 使得python2中也可以使用print函数输出形式
2. 在python2 中， $23/6=3$ ，python中 $23/6=3.8333333333333335$
3. 在python2中使用入 绝对引入 特性。python2中只有 相对引入，比如，文件结构如下：

```
pkg/
pkg/init.py
pkg/main.py
pkg/string.py
```

- 如果我们在main.py中使用 `import string`，在python2.4之前会先查找当前目录下的string模块，但如果 我们想使用系统中的string模块就没有办法了，因为没有好的办法先忽略到同目录下的string。
- 但使用 绝对引用 后，`import string` 导入的是系统默认的模块
- `from pkg import string` 才导入当前目录下的string.py 待考证待考证待考证。。。。。。

1.10 输入与输出

1.10.1 输出格式美化

Python两种输出值的方式: 表达式语句和 print() 函数。

- 第三种方式是使用文件对象的 write() 方法, 标准输出文件可以用 sys.stdout 引用。
- 如果你希望输出的形式更加多样, 可以使用 str.format() 函数来格式化输出值。
- 如果你希望将输出的值转成字符串, 可以使用 repr() 或 str() 函数来实现。
 - str(): 函数返回一个用户易读的表达形式。
 - repr(): 产生一个解释器易读的表达形式。

```
>>> s = 'Hello, Runoob'
>>> str(s)
'Hello, Runoob'
>>> repr(s)
"'Hello, Runoob'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x 的值为: ' + repr(x) + ', y 的值为: ' + repr(y) + '...'
>>> print(s)
x 的值为: 32.5, y 的值为: 40000...
>>> # repr() 函数可以转义字符串中的特殊字符
... hello = 'hello, runoob\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, runoob\n'
>>> # repr() 的参数可以是 Python 的任何对象
... repr((x, y, ('Google', 'Runoob')))
"(32.5, 40000, ('Google', 'Runoob'))"
```

这里有两种方式输出一个平方与立方的表:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))
...
1    1    1
2    4    8
3    9   27

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1    1    1
2    4    8
```

注意：在第一个例子中，每列间的空格由 print() 添加。

- 这个例子展示了字符串对象的 rjust() 方法，它可以将字符串靠右，并在左边填充空格。
- 还有类似的方法，如 ljust() 和 center()。这些方法并不会写任何东西，它们仅仅返回新的字符串。
- 另一个方法 zfill()，它会在数字的左边填充 0，如下所示：

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

str.format() 的基本使用如下：

```
>>> print('{}网址： "{}"'.format('菜鸟教程', 'www.rungoob.com'))
菜鸟教程网址： "www.rungoob.com!"
```

- 括号及其里面的字符（称作格式化字段）将会被 format() 中的参数替换。
- 在括号中的数字用于指向传入对象在 format() 中的位置，如下所示：

```
>>> print('{0} 和 {1}'.format('Google', 'Runoob'))
Google 和 Runoob
>>> print('{1} 和 {0}'.format('Google', 'Runoob'))
Runoob 和 Google
```

- 如果在 format() 中使用了关键字参数，那么它们的值会指向使用该名字的参数。

```
>>> print('{name}网址： {site}'.format(name='菜鸟教程', site='www.rungoob.com'))
菜鸟教程网址： www.rungoob.com
```

- 位置及关键字参数可以任意的结合：

```
>>> print('站点列表 {0}, {1}, 和 {other}' .format('Google', 'Runoob',
                                                 other='Taobao'))
站点列表 Google, Runoob, 和 Taobao。
```

- '!a'（使用 ascii()），'!s'（使用 str()）和 '!r'（使用 repr()）可以用于在格式化某个值之前对其进行转化：

```
>>> import math
>>> print('常量 PI 的值近似为： {}' .format(math.pi))
常量 PI 的值近似为： 3.141592653589793。
>>> print('常量 PI 的值近似为： {!r}' .format(math.pi))
常量 PI 的值近似为： 3.141592653589793。
```

- 可选项 ':' 和格式标识符可以跟着字段名。这就允许对值进行更好的格式化。下面的例子将 Pi 保留到小数点后三位：

```
>>> import math  
>>> print('常量 PI 的值近似为 {:.3f}。'.format(math.pi))  
常量 PI 的值近似为 3.142。
```

- 在 ':' 后传入一个整数，可以保证该域至少有这么多的宽度。用于美化表格时很有用。

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}  
>>> for name, number in table.items():  
...     print('{0:10} ==> {1:10d}'.format(name, number))  
...  
Runoob      ==>      2  
Taobao      ==>      3  
Google      ==>      1
```

- 如果你有一个很长的格式化字符串，而你不想将它们分开，那么在格式化时通过变量名而非位置会是很好的事情。最简单的就是传入一个字典，然后使用方括号 '[]' 来访问键值：

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}  
>>> print('Runoob: {0[Runoob]:d}; Google: {0[Google]:d}; Taobao:  
{0[Taobao]:d}'.format(table))  
Runoob: 2; Google: 1; Taobao: 3
```

- 也可以通过在 table 变量前使用 '**' 来实现相同的功能：(加了两个星号的参数会以字典的形式导入-详见函数的参数传递)

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}  
>>> print('Runoob: {Runoob:d}; Google: {Google:d}; Taobao: {Taobao:d}'.format(**table))  
Runoob: 2; Google: 1; Taobao: 3
```

- % 操作符也可以实现字符串格式化。它将左边的参数作为类似 sprintf() 式的格式化字符串，而将右边的代入，然后返回格式化后的字符串。例如：

```
>>> import math  
>>> print('常量 PI 的值近似为: %5.3f。' % math.pi)  
常量 PI 的值近似为: 3.142。
```

因为 str.format() 比较新的函数，大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除，应该更多的使用 str.format()。

1.10.2 读取键盘输入

Python 提供了 input() 内置函数从标准输入读入一行文本，默认的标准输入是键盘。

input 可以接收一个 Python 表达式作为输入，并将运算结果返回。

```
#!/usr/bin/python3

str = input("请输入：");
print ("你输入的内容是：", str)
```

这会产生如下的对应着输入的结果：

```
请输入：菜鸟教程
你输入的内容是： 菜鸟教程
```

1.11 文件与系统

1.11.1 open() 方法

Python open() 方法用于打开一个文件，并返回文件对象，在对文件进行处理过程都需要使用到这个函数，如果该文件无法被打开，会抛出 OSError。

注意：使用 open() 方法一定要保证关闭文件对象，即调用 close() 方法。

open() 函数常用形式是接收两个参数：文件名(file)和模式(mode)。

```
open(file, mode='r')
```

完整的语法格式为：

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True,
opener=None)
```

参数说明:

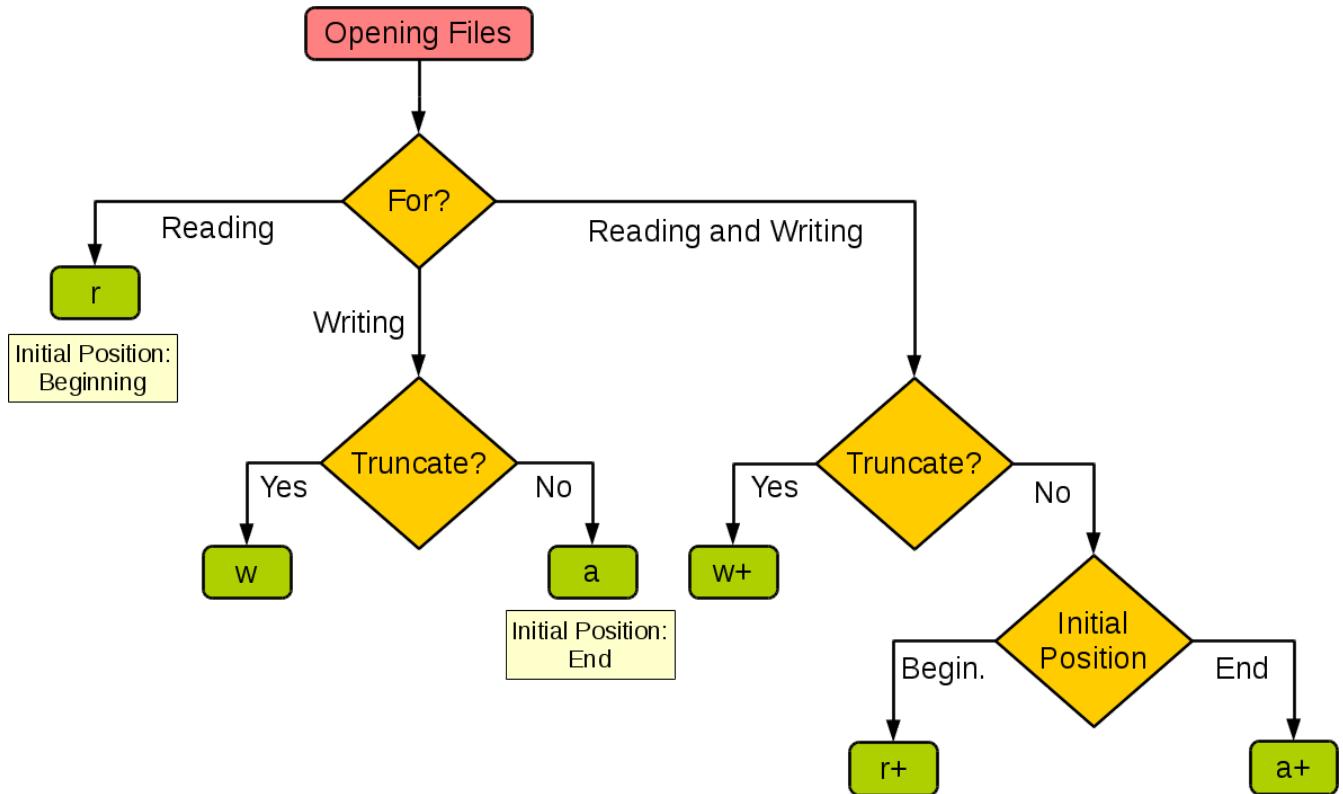
- file: 必需，文件路径（相对或者绝对路径）。
- mode: 可选，文件打开模式
- buffering: 设置缓冲
- encoding: 一般使用utf8
- errors: 报错级别
- newline: 区分换行符
- closefd: 传入的file参数类型
- opener:

mode 参数有：

模式	描述
t	文本模式(默认)。
x	写模式，新建一个文件，如果该文件已存在则会报错。
b	二进制模式。
+	打开一个文件进行更新(可读可写)。
U	通用换行模式(不推荐)。
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
w+	打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

默认为文本模式，如果要以二进制模式打开，加上 b。

下图很好的总结了这几种模式：



模式	r	r+	w	w+	a	a+
读	+	+		+		+
写		+	+	+	+	+
创建			+	+	+	+
覆盖(Truncate)			+	+		
指针在开始	+	+	+	+		
指针在结尾					+	+

以下实例将字符串写入到文件 foo.txt 中：

```

#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

f.write( "Python 是一个非常好的语言。\\n是的，的确非常好!!\\n" )

# 关闭打开的文件
f.close()
  
```

- 第一个参数为要打开的文件名。
- 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选

的; 'r' 将是默认值。

此时打开文件 foo.txt, 显示如下:

```
$ cat /tmp/foo.txt
Python 是一个非常好的语言。
是的，的确非常好！！
```

1.11.2 file 对象

file 对象使用 open 函数来创建, 下表列出了 file 对象常用的函数:

序号	方法及描述
1	file.close() 关闭文件。关闭后文件不能再进行读写操作。
2	file.flush() 刷新文件内部缓冲, 直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。
3	file.fileno() 返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如os模块的read方法等一些底层操作上。
4	file.isatty() 如果文件连接到一个终端设备返回 True, 否则返回 False。
5	file.next() 返回文件下一行。
6	[file.read(size)]从文件读取指定的字节数, 如果未给定或为负则读取所有。
7	[file.readline(size)]读取整行, 包括 "\n" 字符。
8	[file.readlines(sizeint)]读取所有行并返回列表, 若给定sizeint>0, 返回总和大约为sizeint字节的行, 实际读取值可能比 sizeint 较大, 因为需要填充缓冲区。
9	[file.seek(offset , whence)]设置文件当前位置
10	file.tell() 返回文件当前位置。
11	[file.truncate(size)]从文件的首行首字符开始截断, 截断文件为 size 个字符, 无 size 表示从当前位置截断; 截断之后后面的所有字符被删除, 其中 Widnows 系统下的换行代表2个字符大小。
12	file.write(str) 将字符串写入文件, 返回的是写入的字符长度。
13	file.writelines(sequence) 向文件写入一个序列字符串列表, 如果需要换行则要自己加入每行的换行符。

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容, 调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

- size 是一个可选的数字类型的参数。当 size 被忽略了或者为负,那么该文件的所有内容都将被读取并且返回。

以下实例假定文件 foo.txt 已存在（上面实例中已创建）：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.read()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
是的，的确非常好！！
```

f.readline()

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串,说明已经已经读取到最后一行。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readline()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
```

f.readlines()

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readlines() # 返回一个包含换行符的列表
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
['Python 是一个非常好的语言。\\n', '是的，的确非常好！！\\n']
```

- 另一种方式是迭代一个文件对象然后读取每行：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

for line in f:
    print(line, end='')

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
是的，的确非常好！！
```

这个方法很简单，但是并没有提供一个很好的控制。因为两者的处理机制不同，最好不要混用。

f.write()

- f.write(string) 将 string 写入到文件中，然后返回写入的字符数。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

num = f.write("Python 是一个非常好的语言。\\n是的，的确非常好!!\\n")
print(num)
# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
29
```

- 如果要写入一些不是字符串的东西，那么将需要先进行转换：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo1.txt", "w")

value = ('www.runoob.com', 14)
s = str(value)
f.write(s)

# 关闭打开的文件
f.close()
```

执行以上程序，打开 foo1.txt 文件：

```
$ cat /tmp/foo1.txt
('www.runoob.com', 14)
```

f.tell()

f.tell() 返回文件对象当前所处的位置，它是从文件开头开始算起的字节数。

f.seek()

如果要改变文件当前的位置，可以使用 f.seek(offset, from_what) 函数。

from_what 的值，如果是 0 表示开头，如果是 1 表示当前位置，2 表示文件的结尾，例如：

- seek(x,0)：从起始位置即文件首行首字符开始移动 x 个字符

- `seek(x,1)`： 表示从当前位置往后移动x个字符
- `seek(-x,2)`： 表示从文件的结尾往前移动x个字符

`from_what` 值为默认为0，即文件开头。下面给出一个完整的例子：

```
>>> f = open('/tmp/foo.txt', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # 移动到文件的第六个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节
13
>>> f.read(1)
b'd'
```

f.close()

在文本文件中(那些打开文件的模式下没有b的)，只会相对于文件起始位置进行定位。

当你处理完一个文件后，调用`f.close()`来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

- 当处理一个文件对象时，使用`with`关键字是非常好的方式。在结束后，它会帮你正确的关闭文件。而且写起来也比`try - finally`语句块要简短：

```
>>> with open('/tmp/foo.txt', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有其他方法，如`isatty()`和`truncate()`，但这些通常比较少用。

常用例子

```
#!/usr/bin/env python

with open('/tmp/foo.txt', 'r') as f:
    for line in f:
        ...
```

1.11.3 pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

- 通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。
- 通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象，就能对 file 以读取的形式打开：

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的python对象。

file: 类文件对象，有read()和readline()接口。

实例1：

```
#!/usr/bin/python3
import pickle

# 使用pickle模块将数据对象保存到文件
data1 = {'a': [1, 2.0, 3, 4+6j],
          'b': ('string', u'Unicode string'),
          'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

实例2：

```
#!/usr/bin/python3
import pprint, pickle

#使用pickle模块从文件中重构python对象
pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

1.11.4 OS 文件/目录方法

os 模块提供了非常丰富的方法用来处理文件和目录。常用的方法如下表所示：

获取脚本所在目录

有os.getcwd(), sys.path[0] 和file三种方法，他们的区别如下：

假设目录结构是：

```
C:\test
|-getpath
  |-path.py
  |-sub
    |-sub_path.py
```

然后我们在C:\test下面执行 `python getpath/path.py`，这时 `sub_path.py` (path.py调用sub_path.py)里面与各种用法对应的值其实是：

- `os.getcwd()` "C:\test"，取的是起始执行目录
- `sys.path[0]` "C:\test\getpath"，取的是被初始执行的脚本的所在目录
- `os.path.split(os.path.realpath(__file__))[0]` "C:\test\getpath\sub"，取的是file所在文件sub_path.py的所在目录

遍历文件

1. `os.walk`可以用于遍历指定文件下所有的子目录、非目录子文件。

```
import os
filePath = 'C:\myLearning\pythonLearning201712\carComments\01\' 
for i,j,k in os.walk(filePath):
    print(i,j,k)
```

(2)[os.listdir\(path\)](#) 返回path指定的文件夹包含的文件或文件夹的名字的列表, 这个列表按字母顺序排序。

```
import os
filePath = 'C:\\\\myLearning\\\\pythonLearning201712\\\\carComments\\\\01\\\\'
os.listdir(filePath)
```

创建文件夹

[os.makedirs\(path, mode\)](#) 递归文件夹创建函数。像mkdir(), 但创建的所有intermediate-level文件夹需要包含子文件夹。

```
if not os.path.exists(args.image_folder):
    os.makedirs(args.image_folder)
```

序号	方法及描述
1	os.access(path, mode) 检验权限模式
2	os.chdir(path) 改变当前工作目录
3	os.chflags(path, flags) 设置路径的标记为数字标记。
4	os.chmod(path, mode) 更改权限
5	os.chown(path, uid, gid) 更改文件所有者
6	os.chroot(path) 改变当前进程的根目录
7	os.close(fd) 关闭文件描述符 fd
8	os.closerange(fd_low, fd_high) 关闭所有文件描述符，从 fd_low (包含) 到 fd_high (不包含), 错误会忽略
9	os.dup(fd) 复制文件描述符 fd
10	os.dup2(fd, fd2) 将一个文件描述符 fd 复制到另一个 fd2
11	os.fchdir(fd) 通过文件描述符改变当前工作目录
12	os.fchmod(fd, mode) 改变一个文件的访问权限，该文件由参数fd指定，参数mode是Unix下的文件访问权限。
13	os.fchown(fd, uid, gid) 修改一个文件的所有权，这个函数修改一个文件的用户ID和用户组ID，该文件由文件描述符fd指定。
14	os.fdatasync(fd) 强制将文件写入磁盘，该文件由文件描述符fd指定，但是不强制更新文件的状态信息。
15	os.fdopen(fd[, mode[, bufsize]]) 通过文件描述符 fd 创建一个文件对象，并返回这个文件对象
16	os.fpathconf(fd, name) 返回一个打开的文件的系统配置信息。name为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定（POSIX.1, Unix 95, Unix 98, 和其它）。
17	os.fstat(fd) 返回文件描述符fd的状态，像stat()。
18	os.fstatvfs(fd) 返回包含文件描述符fd的文件的文件系统的信息，像 statvfs()
19	os.fsync(fd) 强制将文件描述符为fd的文件写入硬盘。
20	os.ftruncate(fd, length) 裁剪文件描述符fd对应的文件, 所以它最大不能超过文件大小。
21	os.getcwd() 返回当前工作目录
22	os.getcwdu() 返回一个当前工作目录的Unicode对象
23	os.isatty(fd) 如果文件描述符fd是打开的，同时与tty(-like)设备相连，则返回true, 否则False。
24	os.lchflags(path, flags) 设置路径的标记为数字标记，类似 chflags(), 但是没有软链接
25	os.lchmod(path, mode) 修改连接文件权限
26	os.lchown(path, uid, gid) 更改文件所有者，类似 chown，但是不追踪链接。

序号	方法及描述
27	os.link(src, dst) 创建硬链接，名为参数 dst，指向参数 src
28	
29	os.lseek(fd, pos, how) 设置文件描述符 fd 当前位置为 pos, how 方式修改: SEEK_SET 或者 0 设置从文件开始的计算的 pos; SEEK_CUR 或者 1 则从当前位置计算; os.SEEK_END 或者 2 则从文件尾部开始。在 unix, Windows 中有效
30	os.lstat(path) 像 stat(), 但是没有软链接
31	os.major(device) 从原始的设备号中提取设备 major 号码 (使用 stat 中的 st_dev 或者 st_rdev field)。
32	os.makedev(major, minor) 以 major 和 minor 设备号组成一个原始设备号
33	
34	os.minor(device) 从原始的设备号中提取设备 minor 号码 (使用 stat 中的 st_dev 或者 st_rdev field)。
35	[os.mkdir(path, mode)] 以数字 mode 的 mode 创建一个名为 path 的文件夹。默认的 mode 是 0777 (八进制)。
36	[os.mkfifo(path, mode)] 创建命名管道，mode 为数字，默认为 0666 (八进制)
37	[os.mknod(filename, mode=0600, device)] 创建一个名为 filename 文件系统节点 (文件, 设备特别文件或者命名 pipe)。
38	[os.open(file, flags, mode)] 打开一个文件，并且设置需要的打开选项，mode 参数是可选的
39	os.openpty() 打开一个新的伪终端对。返回 pty 和 tty 的文件描述符。
40	os.pathconf(path, name) 返回相关文件的系统配置信息。
41	os.pipe() 创建一个管道。返回一对文件描述符(r, w) 分别为读和写
42	os.popen(command[, mode[, bufsize]]) 从一个 command 打开一个管道
43	os.read(fd, n) 从文件描述符 fd 中读取最多 n 个字节，返回包含读取字节的字符串，文件描述符 fd 对应文件已达到结尾，返回一个空字符串。
44	os.readlink(path) 返回软链接所指向的文件
45	os.remove(path) 删除路径为 path 的文件。如果 path 是一个文件夹，将抛出 OSError；查看下面的 rmdir() 删除一个 directory。
46	os.removedirs(path) 递归删除目录。
47	os.rename(src, dst) 重命名文件或目录，从 src 到 dst
48	os.renames(old, new) 递归地对目录进行更名，也可以对文件进行更名。
49	os.rmdir(path) 删除 path 指定的空目录，如果目录非空，则抛出一个 OSError 异常。
50	os.stat(path) 获取 path 指定的路径的信息，功能等同于 C API 中的 stat() 系统调用。

序号	方法及描述
51	[os.stat_float_times(newvalue)] 决定stat_result是否以float对象显示时间戳
52	os.statvfs(path) 获取指定路径的文件系统统计信息
53	os.symlink(src, dst) 创建一个软链接
54	os.tcgetpgrp(fd) 返回与终端fd（一个由os.open()返回的打开的文件描述符）关联的进程组
55	os.tcsetpgrp(fd, pg) 设置与终端fd（一个由os.open()返回的打开的文件描述符）关联的进程组为pg。
56	os.tempnam([dir[, prefix]]) Python3 中已删除 。返回唯一的路径名用于创建临时文件。
57	os.tmpfile() Python3 中已删除 。返回一个打开的模式为(w+b)的文件对象。这文件对象没有文件夹入口，没有文件描述符，将会自动删除。
58	os.tmpnam() Python3 中已删除 。为创建一个临时文件返回一个唯一的路径
59	os.ttyname(fd) 返回一个字符串，它表示与文件描述符fd关联的终端设备。如果fd没有与终端设备关联，则引发一个异常。
60	os.unlink(path) 删除文件路径
61	os.utime(path, times) 返回指定的path文件的访问和修改的时间。
62	[os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])](http://www.runoob.com/python3/python3-os-walk.html) 输出在文件夹中的文件名通过在树中游走，向上或者向下。
63	os.write(fd, str) 写入字符串到文件描述符 fd 中。返回实际写入的字符串长度
64	os.path 模块 获取文件的属性信息。

1.12 错误和异常

Python有两种错误很容易辨认：语法错误和异常。

1.12.1 语法错误

```
>>>while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
          ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号 (:) 。

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

1.12.2 异常

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里：

```
>>>10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

异常以不同的类型出现，这些类型都作为信息的一部分打印出来：例子中的类型有 ZeroDivisionError，NameError 和 TypeError。

错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息

异常处理

以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

```
>>>while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again ")
```

try语句按照如下方式工作；

- 首先，执行try子句（在关键字try和关键字except之间的语句）
- 如果没有异常发生，忽略except子句，try子句执行后结束。
- 如果在执行try子句的过程中发生了异常，那么try子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的except子句将被执行。最后执行 try 语句之后的代码。
- 如果一个异常没有与任何的except匹配，那么这个异常将会传递给上层的try中。

一个 try 语句可能包含多个except子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。

处理程序将只针对对应的try子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。

一个except子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如：

```
except (RuntimeError, TypeError, NameError):
    pass
```

最后一个except子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try except 语句还有一个可选的else子句，如果使用这个子句，那么必须放在所有的except子句之后。这个子句将在try子句没有发生任何异常的时候执行。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else 子句比把所有的语句都放在 try 子句里面要好，这样可以避免一些意想不到的、而except又没有捕获的异常。

异常处理并不仅仅处理那些直接发生在try子句中的异常，而且还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常。例如：

```
>>>def this_fails():
    x = 1/0

>>> try:
    this_fails()
except ZeroDivisionError as err:
    print('Handling run-time error:', err)

Handling run-time error: int division or modulo by zero
```

抛出异常

Python 使用 raise 语句抛出一个指定的异常。例如：

```
>>>raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

```
>>>try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise

An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

用户自定义异常

你可以通过创建一个新的异常类来拥有自己的异常。异常类继承自 Exception 类，可以直接继承，或者间接继承，例如：

```
>>>class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__():
        return repr(self.value)

>>> try:
    raise MyError(2*2)
except MyError as e:
    print('My exception occurred, value:', e.value)

My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，类 Exception 默认的 `init()` 被覆盖。

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
```

```

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message

```

定义清理行为

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。例如：

```

>>>try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt

```

以上例子不管 try 子句里面有没有发生异常，finally 子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）：

```

>>>def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:

```

```

        print("division by zero!")
else:
    print("result is", result)
finally:
    print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

预定义的清理行为

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

下面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上：

```

for line in open("myfile.txt"):
    print(line, end="")
#以上这段代码的问题是，当执行完毕后，文件会保持打开状态，并没有被关闭。

#关键词 with 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法：
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
#以上这段代码执行完毕后，就算在处理过程中出问题了，文件 f 总是会关闭。

```

1.13 面向对象

- 类(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 方法:** 类中定义的函数。
- 类变量:** 类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员:** 类变量或者实例变量用于处理类及其实例对象的相关数据。
- 方法重写:** 如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
- 局部变量:** 定义在方法中的变量，只作用于当前实例的类。
- 实例变量:** 在类的声明中，属性是用变量来表示的。这种变量就称为实例变量，是在类声明的内部但是在类的其他成员方法之外声明的。

- **继承**: 即一个派生类 (derived class) 继承基类 (base class) 的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如, 有这样一个设计: 一个Dog类型的对象派生自Animal类, 这是模拟"是一个 (is-a)"关系 (例图, Dog是一个Animal)。
- **实例化**: 创建一个类的实例, 类的具体对象。
- **对象**: 通过类定义的数据结构实例。对象包括两个数据成员 (类变量和实例变量) 和方法。

和其它编程语言相比, Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python中的类提供了面向对象编程的所有基本功能: 类的继承机制允许多个基类, 派生类可以覆盖基类中的任何方法, 方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

1.13.1 类定义

语法格式

```
class ClassName:  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```

类实例化后, 可以使用其属性, 实际上, 创建一个类之后, 可以通过类名访问其属性。

1.13.2 类对象

类对象支持两种操作: 属性引用和实例化。

属性引用使用和 Python 中所有的属性引用一样的标准语法: **obj.name**。

类对象创建后, 类命名空间中所有的命名都是有效属性名。所以如果类定义是这样:

```
#!/usr/bin/python3  
  
class MyClass:  
    """一个简单的类实例"""  
    i = 12345  
    def f(self):  
        return 'hello world'  
  
# 实例化类  
x = MyClass()  
  
# 访问类的属性和方法  
print("MyClass 类的属性 i 为: ", x.i)  
print("MyClass 类的方法 f 输出为: ", x.f())
```

以上创建了一个新的类实例并将该对象赋给局部变量 x, x 为空的对象。

类有一个名为 `__init__()` 的特殊方法（**构造方法**），该方法在类实例化时会自动调用，像下面这样：

```
def __init__(self):
    self.data = []
```

类定义了 `__init__()` 方法，类的实例化操作会自动调用 `__init__()` 方法。如下实例化类 `MyClass`，对应的 `__init__()` 方法就会被调用：

```
x = MyClass()
```

当然，`__init__()` 方法可以有参数，参数通过 `__init__()` 传递到类的实例化操作上。例如：

```
#!/usr/bin/python3

class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
x = Complex(3.0, -4.5)
print(x.r, x.i) # 输出结果: 3.0 -4.5
```

self代表类的实例，而非类

- 类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称，按照惯例它的名称是 `self`。

```
class Test:
    def prt(self):
        print(self)
        print(self.__class__)

t = Test()
t.prt()
```

以上实例执行结果为：

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

- 从执行结果可以很明显的看出，`self` 代表的是类的实例，代表当前对象的地址，而 `self.class` 则指向类。`self` 不是 python 关键字，我们把他换成 `runoob` 也是可以正常执行的

1.13.3 类的方法

在类的内部，使用 `def` 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 `self`，且为第一个参数，`self` 代表的是类的实例。

```
#!/usr/bin/python3
```

```

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性, 私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):
        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说: 我 %d 岁。" %(self.name,self.age))

# 实例化类
p = people('runoob',10,30)
p.speak()

```

1.13.4 继承

Python 同样支持类的继承，如果一种语言不支持继承，类就没有什么意义。派生类的定义如下所示：

```

class DerivedClassName(BaseClassName1):
    <statement-1>
    .
    .
    .
    <statement-N>

```

- 需要注意圆括号中基类的顺序，若是基类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找基类中是否包含方法。
- BaseClassName（示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用：

```
class DerivedClassName(modname.BaseClassName):
```

```

#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性, 私有属性在类外部无法直接进行访问

```

```

__weight = 0
#定义构造方法
def __init__(self,n,a,w):
    self.name = n
    self.age = a
    self.__weight = w
def speak(self):
    print("%s 说：我 %d 岁。" %(self.name,self.age))

#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构函
        people.__init__(self,n,a,w)
        self.grade = g
    #覆写父类的方法
    def speak(self):
        print("%s 说：我 %d 岁了，我在读 %d 年级"%(self.name,self.age,self.grade))

s = student('ken',10,60,3)
s.speak()

```

1.13.5 多继承

Python同样有限的支持多继承形式。多继承的类定义形如下例：

```

class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>

```

需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找父类中是否包含方法。

```

#!/usr/bin/python3

#类定义
class people:
    #定义基本属性
    name = ''
    age = 0
    #定义私有属性，私有属性在类外部无法直接进行访问
    __weight = 0
    #定义构造方法
    def __init__(self,n,a,w):

```

```

        self.name = n
        self.age = a
        self.__weight = w
    def speak(self):
        print("%s 说：我 %d 岁。" %(self.name, self.age))

#单继承示例
class student(people):
    grade = ''
    def __init__(self,n,a,w,g):
        #调用父类的构造函数
        people.__init__(self,n,a,w)
        self.grade = g
    #覆盖父类的方法
    def speak(self):
        print("%s 说：我 %d 岁了，我在读 %d 年级" %(self.name, self.age, self.grade))

#另一个类，多重继承之前的准备
class speaker():
    topic = ''
    name = ''
    def __init__(self,n,t):
        self.name = n
        self.topic = t
    def speak(self):
        print("我叫 %s，我是一个演说家，我演讲的主题是 %s" %(self.name, self.topic))

#多重继承
class sample(speaker,student):
    a =''
    def __init__(self,n,a,w,g,t):
        student.__init__(self,n,a,w,g)
        speaker.__init__(self,n,t)

test = sample("Tim",25,80,4,"Python")
test.speak()  #方法名同，默认调用的是在括号中排前地父类的方法

```

1.13.6 方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法，实例如下：

```
#!/usr/bin/python3

class Parent:          # 定义父类
    def myMethod(self):
        print ('调用父类方法')

class Child(Parent): # 定义子类
    def myMethod(self):
        print ('调用子类方法')

c = Child()           # 子类实例
c.myMethod()          # 子类调用重写方法
super(Child,c).myMethod() #用子类对象调用父类已被覆盖的方法
```

[super\(\)](#)函数是用于调用父类(超类)的一个方法。

1.13.7 类属性与方法

类的私有属性

[__private_attrs](#)：两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。在类内部的方法中使用时 `self.__private_attrs`。

类的方法

在类的内部，使用 `def` 关键字来定义一个方法，与一般函数定义不同，**类方法必须包含参数 self**，且为第一个参数，`self` 代表的是类的实例。

`self` 的名字并不是规定死的，也可以使用 `this`，但是最好还是按照约定是用 `self`。

类的私有方法

[__private_method](#)：两个下划线开头，声明该方法为私有方法，只能在类的内部调用，不能在类的外部调用。
`self.__private_methods`。

```
class JustCounter:
    __secretCount = 0  # 私有变量
    publicCount = 0    # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print (self.__secretCount)

counter = JustCounter()
counter.count() # 1
counter.count() # 2
print (counter.publicCount) # 2
print (counter.__secretCount) # 报错，实例不能访问私有变量
```

```

#!/usr/bin/python3

class Site:
    def __init__(self, name, url):
        self.name = name      # public
        self.__url = url      # private

    def who(self):
        print('name : ', self.name)
        print('url : ', self.__url)

    def __foo(self):          # 私有方法
        print('这是私有方法')

    def foo(self):            # 公共方法
        print('这是公共方法')
        self.__foo()

x = Site('菜鸟教程', 'www.runoob.com')
x.who()      # 正常输出
x.foo()      # 正常输出
x.__foo()    # 报错，外部不能调用私有方法

```

类的专有方法：

- `__init__`：构造函数，在生成对象时调用
- `__del__`：析构函数，释放对象时使用
- `__repr__`：打印，转换
- `__setitem__`：按照索引赋值
- `__getitem__`：按照索引获取值
- `__len__`：获得长度
- `__cmp__`：比较运算
- `__call__`：函数调用
- `__add__`：加运算
- `__sub__`：减运算
- `__mul__`：乘运算
- `__truediv__`：除运算
- `__mod__`：求余运算
- `__pow__`：乘方

运算符重载

Python同样支持运算符重载，我们可以对类的专有方法进行重载，实例如下：

```

#!/usr/bin/python3

class Vector:

```

```

def __init__(self, a, b):
    self.a = a
    self.b = b

def __str__(self):
    return 'Vector (%d, %d)' % (self.a, self.b)

def __add__(self, other):
    return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)

---
Vector(7,8)

```

1.13.8 变量总结

定义

- 全局变量
在模块内、在所有函数外面、在class外面，这就是全局变量
- 局部变量
在函数内、在class的方法内（未加self修饰），这就是局部变量
- 类变量(静态变量)
在class内的，但不在class的方法内的，这就是类变量(静态变量)
- 实例变量
在class的方法内的，用self修饰的变量，这就是实例变量

用法

全局变量

全局变量供全局共享，全局类和函数均可访问，达到同步作用。同时还可以被外部文件访问。

- 全局变量使用的时候，需要用global显示声明
- 一般情况下，如果 **函数** 直接调用全局变量，不做更新的话，一般没有问题，但如果有重新赋值，又没有在函数内部使用global声明的话，就相当于在内部创建了一个同名的局部变量，局部变量优先级要高于全局变量。

```

# coding:utf-8
_all = 'HelloWorld'      # 全局变量
_all_list = []      # 全局变量

def printall():
    global _all      # 这里需要用global显示声明
    print _all      # HelloWorld

```

错误：

```
# coding:utf-8
_all = 0      # 全局变量
_all_list = []     # 全局变量

def printall():
    # 这里不使用global显示声明
    # 更新一下_all，这里其实是printall的局部变量了
    _all = 3
    print _all

printall()      # 结果3
print _all      # 结果为0
```

局部变量

生命期间函数内

```
# coding:utf-8
def show():
    city = 'Beijing'      # 局部变量
    print city
```

类变量

- 可以称为是静态变量，**通过类名可以直接访问**，也可以通过实例名直接访问
- 此变量在类中全局共享，实例之间也是全局共享。可以看下面实例

```
# coding:utf-8
class foo:
    all = 0
    b = 0
    def add(self):
        foo.all += 1 # 通过类名访问类变量，此变量在类内全局共享
        b +=1 # 错误！！这里b是局部变量，没有定义！

instance01 = foo()    # 实例化对象1
instance02= foo()    # 实例化对象2
print instance01.all    # 通过实例名直接访问，执行结果为：0
print instance02.all    # 通过实例名直接访问，执行结果为：0
print foo.all    # 通过类名直接访问，执行结果为：0

instance01.add()
print instance01.all    # 执行结果为：1
print instance02.all    # 执行结果为：1
print foo.all    # 执行结果为：1
# 类变量在实例之间也全局共享

instance02.add()
print instance01.all    # 执行结果为：2
```

```
print instance02.all    # 执行结果为: 2
print foo.all    # 执行结果为: 2
```

实例变量

- 对于模块来说，有了自己的全局变量，可以供自己内部的类，函数使用，同步；
- 对于函数或者类方法来说，有了自己的局部变量，供自己内部使用；
- 对于类，有了类变量，可以供内部和有继承关系的父子之间使用，同步；
- 但实例之间各自的私有变量（局部变量）就要靠实例变量了，实现了动态绑定，多态特性。
用self来修饰

```
# coding:utf-8
class foo:
    all = 0
    def __init__(self, name):
        self.name = name
    def add(self):
        foo.all += 1 # 类变量实现了类内类外共享
        b = 0 # 局部变量只能在此方法内使用

# name 仅和自己的实例绑定
instance01 = foo('hello')
instance02 = foo('hi')
print instance01.name    # hello
print instance02.name    # hi
print foo.name    # AttributeError: class foo has no attribute 'name'
```

总结

在整个代码组织的过程中，一般会有这样两种基本需求，**私有** 和 **公有**。

- 私有：不与其他共享，自己独享，如函数和方法的局部变量，实例变量
- 公有：需要在一定范围内共享，达到同步目的，如模块内的代码共享的全局变量，类与子类之间共享的静态变量

类变量与实例变量解析

- 类变量：

类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。类变量在所有实例化的对象中是公用的。

- 实例变量：

定义在方法中的变量，用 self 绑定到实例上，只作用于当前实例的类。

访问方式

- 类变量：

- 在类内部和外部，类变量都用 `类名.类变量` 的形式访问。

- 在类内部，也可以用 `self.类变量` 来访问类变量，但此时它的含义已经变了，实际上它已经成了一个实例变量。在实例变量没有被重新赋值时，用 `self.类变量` 才能访问到正确的值。简单的说就是实例变量会屏蔽掉类变量的值，就像局部变量屏蔽掉全局变量的值一样。所以一般情况下是不将类变量作为实例变量使用的
- 实例变量：
 - 在类内部，实例变量用 `self.实例变量` 的形式访问；在类外部，用 `实例名.实例变量` 的形式访问。
 - 实例变量是绑定到一个实例上的变量，它只属于这个实例。
 - 这一点区别于类变量，类变量是所有来自于同一个类的实例所共享的。

self 简介

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称，但是在调用这个方法的时候你不为这个参数赋值，Python 会提供这个值。这个特别的变量指对象本身，按照惯例它的名称是 `self`。虽然你可以给这个参数任何名称，但是强烈建议你使用 `self` 这个名称。

实际上，实例变量就是一个用 `self` 修饰的变量。`self` 将一个变量绑定到一个特定的实例上，这样它就属于这实例自己。

举例说明

```
# -*- coding: utf-8 -*-

#       Author @  Huoty
# Create date @  2015-12-19 09:39:36

class A(object):
    va = 10 # 类变量

    def foo(self):
        print A.va #
        print self.va # va变成了实例变量，不修改可以读取，但修改后会屏蔽类变量的值，在obj1.va 与
obj2.va中体现，类内不体现

        self.va = 40
        print A.va
        print self.va

        va = 20 # 局部变量
        print va

        A.va = 15
        print A.va
        print self.va

# Script starts from here

obj1 = A()
obj2 = A()
obj1.foo() # 10, 10,
print A.va
print obj1.va
print obj2.va
```

程序的输出结果：

```
10  
10  
10  
40  
20  
15  
40  
15  
40  
15
```

分析程序结果我们发现，一般情况下，类变量可以用类名或者self的形式访问。当 `self.类变量` 被重新赋值时，它的值就发生了变化，但类变量的值不会随之变化。方法内的局部变量会屏蔽掉类变量和实例变量。类变量是所有实例共享的，而实例变量只属于对象自己，每个实例的实例变量可以有不同的值。

要点总结

- 1、类变量可以用 `类名.类变量` 和 `self.类变量` 两种方式访问，后者一般情况不建议使用。
- 2、类变量是所有对象所共享的，无论任何时候都建议用类名的方式访问类变量。
- 3、实例变量在类内部用 `self` 访问，在类外部用实例名访问。
- 4、类变量通过 `self` 访问时则被转化为实例变量，被绑定到特定的实例上，其值会屏蔽掉类变量。
- 5、通过对实例变量（`self`）的形式对类变量重新赋值后，类变量的值不随之变化。
- 6、方法内的局部变量会屏蔽掉类变量和实例变量。
- 7、同一实例变量在不同的实例中可能拥有不同的值。

1.14 标准库概览

1.14.1 操作系统接口

`os`模块提供了不少与操作系统相关联的函数。

```
>>> import os  
>>> os.getcwd()      # 返回当前的工作目录  
'C:\\Python34'  
>>> os.chdir('/server/accesslogs')    # 修改当前的工作目录  
>>> os.system('mkdir today')    # 执行系统命令 mkdir  
0
```

- 建议使用 `"import os"` 风格而非 `"from os import *"`。这样可以保证随操作系统不同而有所变化的 `os.open()` 不会覆盖内置函数 `open()`。

在使用 `os` 这样的大型模块时内置的 `dir()` 和 `help()` 函数非常有用：

```
>>> import os  
>>> dir(os)  
<returns a list of all module functions>  
>>> help(os)  
<returns an extensive manual page created from the module's docstrings>
```

针对日常的文件和目录管理任务，`:mod:shutil` 模块提供了一个易于使用的高级接口：

```
>>> import shutil  
>>> shutil.copyfile('data.db', 'archive.db')  
>>> shutil.move('/build/executables', 'installldir')
```

1.14.2 文件通配符

`glob`模块提供了一个函数用于从目录通配符搜索中生成文件列表：

```
>>> import glob  
>>> glob.glob('*.*py')  
['primes.py', 'random.py', 'quote.py']
```

1.14.3 命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以列表形式存储于 `sys` 模块的 `argv` 变量。

示例：`demo.py`

```
import sys  
print(sys.argv)  
print('\nPython 路径为：', sys.path, '\n')
```

在命令行中执行 `python3 demo.py one two three` 后可以得到以下输出结果：

```
['demo.py', 'one', 'two', 'three']  
  
Python 路径为： ['/home/ubuntu16/test', '/usr/lib/python35.zip', '/usr/lib/python3.5',  
'/usr/lib/python3.5/plat-x86_64-linux-gnu', '/usr/lib/python3.5/lib-dynload',  
'/home/ubuntu16/.local/lib/python3.5/site-packages', '/usr/local/lib/python3.5/dist-  
packages', '/usr/lib/python3/dist-packages']
```

- `sys.argv` 是一个包含命令行参数的列表。`sys.argv[0]` 是脚本名
- `sys.path` 包含了一个 Python 解释器自动查找所需模块的路径的列表。`sys.path[0]` 是脚本所在位置

命令行解析工具 gflags

`google flags` 是 Google 使用的一个开源库，用于解析命令行标记，有 C++ 和 Python 两个版本，这里介绍 Python 版本。

安装:

```
pip install python-gflags
```

示例:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import sys
import gflags
import logging

# 给 gflags.FLAGS 定义别名 (实例化)
Flags = gflags.FLAGS

# 使用gflags.DEFINE_type定义 参数名, 参数值, 参数说明
gflags.DEFINE_string('name', 'func_test', 'test function name')
gflags.DEFINE_integer('qps', 0, 'test qps')
gflags.DEFINE_boolean('debug', False, 'whether debug')

def main(argv):
    # 对所有参数进行初始化处理
    Flags(argv)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',
                        datefmt='%a, %d %b %Y %H:%M:%S',
                        filename='test.log'
                        filemode='w')

    # 调用时直接通过 . 操作符调用相关参数
    logging.debug(Flags.name)
    logging.info(Flags.qps)
    logging.warning(Flags.debug)

if __name__ == "__main__":
    main(sys.argv)
```

使用:

- 运行时使用 `--参数名 参数值` 对参数赋值

```
python demo.py --name=test --qps=111 --debug=false
```

此时查看根目录，出现了test.log文件，其内容为：

```
Sat, 20 Jul 2019 14:59:53 demo.py[line:26] DEBUG test
Sat, 20 Jul 2019 14:59:53 demo.py[line:27] INFO 111
Sat, 20 Jul 2019 14:59:53 demo.py[line:28] WARNING False
```

- 使用`--help` 查看所有参数的定义

```
python demo.py --help
```

Remark:

- 如果要配置的参数过多，也支持从cfg配置文件中批量读取

C++版本使用：

http://notes.tanchuanqi.com/language/cpp/google_library.html#gflags

<https://izualzhy.cn/gflags-introduction#8-version%E4%B8%8Ehelp>

https://blog.csdn.net/tommy_wxie/article/details/77649064

1.14.4 错误输出重定向和程序终止

sys 还有 stdin, stdout 和 stderr 属性，即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息。

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

大多脚本的定向终止都使用 "sys.exit()"。

1.14.5 字符串正则匹配

re模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正则表达式提供了简洁、优化的解决方案：

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

如果只需要简单的功能，应该首先考虑字符串方法，因为它们非常简单，易于阅读和调试：

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

1.14.6 数学

math模块为浮点运算提供了对底层C函数库的访问：

```
>>> import math  
>>> math.cos(math.pi / 4)  
0.70710678118654757  
>>> math.log(1024, 2)  
10.0
```

random提供了生成随机数的工具。

```
>>> import random  
>>> random.choice(['apple', 'pear', 'banana'])  
'apple'  
>>> random.sample(range(100), 10)    # sampling without replacement  
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]  
>>> random.random()    # random float  
0.17970987693706186  
>>> random.randrange(6)    # random integer chosen from range(6)  
4
```

1.14.7 访问互联网

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 `urllib.request` 以及用于发送电子邮件的 `smtplib`:

```
>>> from urllib.request import urlopen  
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):  
...     line = line.decode('utf-8')    # Decoding the binary data to text.  
...     if 'EST' in line or 'EDT' in line:    # look for Eastern Time  
...         print(line)  
  
<BR>Nov. 25, 09:43:32 PM EST  
  
>>> import smtplib  
>>> server = smtplib.SMTP('localhost')  
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',  
...     """To: jcaesar@example.org  
...     From: soothsayer@example.org  
...  
...     Beware the Ides of March.  
...     """)  
>>> server.quit()
```

注意第二个例子需要本地有一个在运行的邮件服务器。

1.14.8 日期和时间

`datetime`模块为日期和时间处理同时提供了简单和复杂的方法。

支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。

该模块还支持时区处理:

更多细节参见: <https://www.programiz.com/python-programming/datetime/strftime>

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368

# 输出当前时间:
datetime.datetime.now().strftime("%Y_%m_%d_%H_%M_%S_%f")
2019_04_18_15_07_39_206359
```

Code	Meaning	Example
%a	Weekday as locale's abbreviated name.	Mon
%A	Weekday as locale's full name.	Monday
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	1
%d	Day of the month as a zero-padded decimal number.	30
%-d	Day of the month as a decimal number. (Platform specific)	30
%b	Month as locale's abbreviated name.	Sep
%B	Month as locale's full name.	September
%m	Month as a zero-padded decimal number.	09
%-m	Month as a decimal number. (Platform specific)	9
%y	Year without century as a zero-padded decimal number.	13
%Y	Year with century as a decimal number.	2013
%H	Hour (24-hour clock) as a zero-padded decimal number.	07
%-H	Hour (24-hour clock) as a decimal number. (Platform specific)	7
%I	Hour (12-hour clock) as a zero-padded decimal number.	07
%-I	Hour (12-hour clock) as a decimal number. (Platform specific)	7
%p	Locale's equivalent of either AM or PM.	AM
%M	Minute as a zero-padded decimal number.	06
%-M	Minute as a decimal number. (Platform specific)	6
%S	Second as a zero-padded decimal number.	05
%-S	Second as a decimal number. (Platform specific)	5
%f	Microsecond as a decimal number, zero-padded on the left.	000000
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).	
%Z	Time zone name (empty string if the object is naive).	
%j	Day of the year as a zero-padded decimal number.	273
%-j	Day of the year as a decimal number. (Platform specific)	273
%U	Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	
39		
%W	Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0.	39

```
%c Locale's appropriate date and time representation. Mon Sep 30 07:06:05 2013
%x Locale's appropriate date representation. 09/30/13
%X Locale's appropriate time representation. 07:06:05
%% A literal '%' character. %
```

1.14.9 数据压缩

以下模块直接支持通用的数据打包和压缩格式： zlib， gzip， bz2， zipfile， 以及 tarfile。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

1.14.10 性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人得多，timeit 证明了现代的方法更快一些。

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

相对于 timeit 的细粒度， :mod:profile 和 pstats 模块提供了针对更大代码块的时间度量工具。

1.14.11 测试模块

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试

doctest 模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。

测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。

通过用户提供的例子，它强化了文档，允许 doctest 模块确认代码的结果是否与文档一致：

```

def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()  # 自动验证嵌入测试

```

unittest模块不像 doctest模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集：

```

import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests

```

1.15 python2.x 与3.x 版本兼容

The state of Python 3 adoption: <https://rushter.com/blog/python-3-adoption/>

0. 从Python 3开始开发

如果你有了一个新的点子打算用Python来实现，那么你应该使用Python 3的语法进行开发，并确保向下兼容，而不是从Python 2开始向上升级。如果你手里拿到的是“祖传”Python 2代码需要重构，可能更麻烦一些。官方为这种情况提供了一个自动化工具，叫做2to3（[26.7. 2to3 - Automated Python 2 to 3 code translation](#)）用来转换Python 2到Python 3。我用过几次但感觉不大适合非常复杂的项目，尤其是命令行对整个目录使用后不好追踪变化，而且这个工具库并不能处理每一个问题，常常有一些误操作，修复bug也很花时间。所以建议在单个文件上试用，再决定是整体重构还是用2to3。

因为从Python 2重构到3比较复杂，这个回答主要分享一些从Python 3向下兼容或者直接从Python 3开始开发的建议。

1. 选择合理的兼容范围

放弃Python 2.6及以前的版本，只支持Python 2.7。同理，Python 3可以从3.4开始支持，这样难度会小很多。大部分情况下开源社区的约定俗成是支持Python 2.7以及3.4+。原因包括：

- 2.7以前的版本不兼容的部分较多，额外工作量太大且已经停止了更新支持。官方迁移指南中直接写明了放弃Python 2.6及以前版本（[Porting Python 2 Code to Python 3](#)）。
- 使用Python 2.7的人占了绝大多数，从投入产出上比较划算。

2. 善用兼容库six和Python内置的future

six是一个第三方工具库，专门用来兼容Python 2和3。聪明的读者肯定已经发现了 $2*3=6$ ，这也是six这个工具库的名字由来。很多Python 2和3中语法不兼容，或者不存在的部分都可以用six来处理。

举个简单的例子，我们很多情况下都要写继承，也就是使用Python中的abc和abstractmethod。

在Python 3中，使用继承的语法是这样的：

```
from abc import ABC, abstractmethod

class Base(ABC):
    @abstractmethod
    def do_something(self):
        pass
```

但是以上语法会在Python 2下报错，因为Python 2中并没有ABC无法import。那么使用six这个工具库，我们就可以写出兼容代码，如下：

```
import abc
import six

@six.add_metaclass(abc.ABCMeta)
class Base(object):
    @abc.abstractmethod
    def do_something(self):
        pass
```

更多six的例子可以看这里：[Six: Python 2 and 3 Compatibility Library](#)

3. 避开API断层或者使用替代品

有不少关键API在Python 2和3中都有变化，详细的介绍可以看[Porting Python 2 Code to Python 3](#)。就我个人而言，比较常见的坑有：

- 编码（encoding）问题，Python 2的编码问题已经不是个新闻。一般来讲可以通过在文件头加 `# -*- coding: utf-8 -*-` 来强制用utf-8编码。
- 整除问题：python 2中的除法是默认取整的，比如 $2/3=0$ 。而Python 3中就会得到 0.6666667 。这种情况下，可以使用内置的**future****来解决问题，这下即使在Python 2下也可以默认得到浮点结果了。

```
from __future__ import division
```

- 输出、打印问题。`print`函数在Python 2和3下也有所不同，比如括号的问题。这个也可以用内置的**future****来解决问题，这样在python 2下运行`print()`也不会多出额外的一对括号了。

```
from __future__ import print_function
```

- 文件读取中也有一些麻烦，比如Python 3中的打开文件时可以指定文件编码：

```
with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

如果在Python 2下运行这个代码，就会报错提示没有“encoding”这个参数，因为Python 2内置的open () 所使用的参数和Python 3不同。这种情况下可以使用io中的open函数，这样就同时兼容2和3了：

```
from io import open

with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

另一种兼容的方法是调用第三方库，而回避一些Python的内置函数。举个例子，检测两个数字是否接近在Python 3中可以通过math.isclose()这个内置函数来实现，而在Python 2中是不存在这个函数的。这种情况下，我们可以调用numpy来替代python的内置函数，用np.isclose()就回避了Python版本的问题。同理sklearn中也提供了不少用来兼容的包，比如在Python 3中我们常常使用

```
from inspect import signatures
```

signature主要适用于查看函数和类的attribute和methods，但Python 2中的inspect没有signature这个函数。这种情况下，如果你的代码依赖于sklearn，就可以用：

```
from sklearn.externals.funcsigs import signature
```

事实上 sklearn.externals提供了不少有用的兼容性工具，甚至连six都可以直接从sklearn中import

```
from sklearn.externals import six
```

4. 增加判别语句

对于不同的Python版本，可以选择安装不同的工具库。软件要求一般写在了requirements.txt文件中，可以选择根据Python版本安装不同的工具库。下面的内容就告诉了setuptools仅在Python版本低于3时安装pathlib2，而scipy是强制要求的不管是Python 2或者3。格式如下：

工具库名称； python_version < '版本号'

```
scipy
pathlib2 ; python_version < '3' # 仅当python版本低于3时安装
```

在import一个和版本有关的函数时，可以选择加上try except来import合适的库：

```
try:
    from pathlib import Path
except ImportError:
    from pathlib2 import Path # 如果没有pathlib就使用pathlib2
```

如果愿意，以上代码也可以改写成：

```
import sys
if (sys.version_info > (3, 0)):
    # 如果是Python 3
    import pathlib
else:
    # 如果是Python 2
    import pathlib2
```

5. 使用自动测试和持续集成

有很多工具可以帮助你在提交新的代码后自动在不同环境下测试是否可以运行和通过测试。比如travis-ci，你只需要提供想要测试的代码环境即可，这样就可以降低测试多个环境的开销和难度。

比如下图就显示该工具库可以通过3.4以上的版本，但挂在了2.7上，开了自动测试和持续集成可以第一时间发现是否哪个版本出了错误，好迅速解决。

Build Jobs

✗ #	Python: 2.7	no environment variables set
✓ #	Python: 3.4	no environment variables set
✓ #	Python: 3.5	no environment variables set
✓ #	Python: 3.5-dev	no environment variables set
✓ #	Python: 3.6	no environment variables set
✓ #	Python: 3.6-dev	no environment variables set

知乎 @微调

6. 总结

写出同时兼容Python 2和3的代码有很大的价值，因为很多公司和个人还没有做好到Python 3的迁移工作，短时间内Python 2依然是主流。

其实从兼容性上来说难度并不是很大，就是个试错过程（trials and errors）。造轮子已经不易，何不稍下功夫来获得更大的市场呢？

2. Intermediate Tutorial

Numpy

[Numpy](#) is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find [this tutorial useful](#) to get started with Numpy.

Arrays

数组创建

A numpy array is a grid of values, all of the **same type**, and is indexed by a tuple of nonnegative integers. The number of **dimensions is the rank (axis) of the array**; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np

a = np.array([1, 2, 3])      # Create a rank 1 array
print(type(a))              # Prints "<class 'numpy.ndarray'>"
print(a.shape)               # Prints "(3,)"
print(a[0], a[1], a[2])     # Prints "1 2 3"
a[0] = 5                    # Change an element of the array
print(a)                     # Prints "[5, 2, 3]

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])  # Prints "1 2 4"
```

Numpy also provides many functions to create arrays:

```
import numpy as np

a = np.zeros((2,2))      # Create an array of all zeros, 维度参数是一个元组
print(a)                  # Prints "[[ 0.  0.]
                           #           [ 0.  0.]]"

b = np.ones((1,2))       # Create an array of all ones
print(b)                  # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)    # Create a constant array
print(c)                  # Prints "[[ 7.  7.]
                           #           [ 7.  7.]]"

d = np.eye(2)             # Create a 2x2 identity matrix
print(d)                  # Prints "[[ 1.  0.]
                           #           [ 0.  1.]]"

e = np.random.random((2,2)) # Create an array filled with random values
print(e)                  # Might print "[[ 0.91940167  0.08143941]
                           #           [ 0.68744134  0.87236687]]"
```

- You can read about other methods of array creation [in the documentation](#).
- 关于如何高效创建numpy array 参见https://ask.helplib.com/python/post_544244

数组查询

ndim:

返回的是数组的维度，返回的只有一个数，该数即表示数组的维度（轴的个数）。

shape:

返回表示各位维度大小的元组。

- 对于一维数组：是（6，）为什么不是（1，6），因为ndim维度为1，元组内只返回一个数。
- 对于二维数组：按照我们对矩阵的理解，第一个数代表行 轴0，第二个数代表列，轴1
- 对于多维数组：元组内维度的顺序按照数组轴的顺序出现。最后两维度便是我们理解的行和列，在做维度变换时要注意，多维数组也可以理解成对二维的堆叠。

dtype:

返回的是该数组的数据类型。由于图中的数据都为整形，整形返回的都是int32，浮点型就会返回float64。这是numpy使用的数据类型，注意与其他库使用的数据类型区分并判定是否兼容。比如存储图像时，一般使用Int8。

astype

转换数组的数据类型。

int32 --> float64

float64 --> int32 会将小数部分截断

string_ --> float64 如果字符串数组表示的全是数字，也可以用astype转化为数值类型

Array indexing

Numpy offers several ways to index into arrays.

Slicing:

Similar to Python lists, numpy arrays can be sliced.

Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create the following **rank 2 array** with shape (3, 4)
# [[ 1  2  3  4]
# [ 5  6  7  8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
# [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])    # Prints "77"
```

- You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:
- **Noticing the rank of array and the difference of shape caused by two kinds of slice methods!**

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
# [ 5  6  7  8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]      # Rank 1 view of the second row of a
row_r2 = a[1:2, :]    # Rank 2 view of the second row of a
print(row_r1, row_r1.shape) # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape) # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1] # 1 dimension
col_r2 = a[:, 1:2] # two dimension
print(col_r1, col_r1.shape) # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape) # Prints "[[ 2
                            #
                            [ 6]
                            [10]] (3, 1)"
```

Integer array indexing:

- When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array.
- In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]]) # Prints "[1 4 5]",参数看作一个 index array,首先是轴0的index,然后是轴1的index,最后组成三个对应元素的index对

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]])) # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]]) # Prints "[2 2]"
```

```
# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]])) # Prints "[2 2]"
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a) # prints "array([[ 1,  2,  3],
#                  [ 4,  5,  6],
#                  [ 7,  8,  9],
#                  [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b]) # Prints "[ 1  6  7 11]" #np.arange(4)=array([0, 1, 2, 3])

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10 # won't create a new array

print(a) # prints "array([[11,  2,  3],
#                  [ 4, 15, 16],
#                  [17,  8,  9],
#                  [10, 21, 12]])"
```

Boolean array indexing:

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2) # Find the elements of a that are bigger than 2;
# this returns a numpy array of Booleans of the same
# shape as a, where each slot of bool_idx tells
# whether that element of a is > 2.

print(bool_idx) # Prints "[[False False]
#                  [ True  True]
#                  [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx]) # Prints "[3 4 5 6]"
```

```
# We can do all of the above in a single concise statement:  
print(a[a > 2])      # Prints "[3 4 5 6]"
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should [read the documentation](#).

Array Stacking

`row_stack()` & `column_stack()`

- stack one dimensional list or multiple dimensional array along the row or column. The inputs can be numpy arrays or python lists or both of them. But the outputs will always be a numpy array.
- note that the input data type is tuple!

```
import numpy as np  
  
# case1: 1-list + 1-list  
>>> x = [1,2]  
>>> row = [3,4]  
>>> x = np.row_stack((x,row))  
>>> x  
array([[1, 2],  
       [3, 4]])  
  
# case2: 1-list + 2-list  
>>> x = [1,2]  
>>> row2 = [[3,4],[5,6]]  
>>> x = np.row_stack((x,row2))  
>>> x  
array([[1, 2],  
       [3, 4],  
       [5, 6]])  
  
# case3: 1-list + 2-array  
>>> row2 = np.array([[3,4],[5,6]])  
>>> x = np.row_stack((x,row2))  
>>> x  
array([[1, 2],  
       [3, 4],  
       [5, 6]])  
  
# case4: 2-array + 1-list  
>>> x = np.array(([1,2],[3,4]))  
>>> col = [5,6]  
>>> x = np.column_stack((x,col))  
>>> x  
array([[1, 2, 5],  
       [3, 4, 6]])
```

```

# case5: 2-array + 2-list
>>> x = np.array(([1,2],[3,4]))
>>> col = [[5,6],[7,8]]
>>> x = np.column_stack((x,col))
>>> x
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])

# case6: 2-array + 2-array
>>> x = np.array(([1,2],[3,4]))
>>> col = np.array([[5,6],[7,8]])
>>> x = np.column_stack((x,col))
>>> x
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])

# case7: 2-list + 2-list
>>> x = [[1,2],[3,4]]
>>> col = [[5,6],[7,8]]
>>> x = np.row_stack((x,col))
>>> x
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])

```

hstack()、vstack () 、stack () 、dstack () 、vsplit () 、concatenate ()

- stack () : 沿着新的轴加入一系列数组。
- vstack () : 堆栈数组垂直顺序（行）
- hstack () : 堆栈数组水平顺序（列）。
- dstack () : 堆栈数组按顺序深入（沿第三维）。
- vsplit () : 将数组分解成垂直的多个子数组的列表。
- concatenate () : 连接沿现有轴的数组序列。
- 判断stack之后的维度根据原始各个轴的维度以及堆叠的维度来判断，最外面的括号是轴0，依次往里增加；

1. numpy.stack()函数

函数原型：numpy.stack(arrays, axis=0)

示例：

```
In [34]: arrays = [np.random.randn(3,4) for _ in range(10)]
np.stack(arrays, axis=0).shape
```

```
Out[34]: (10L, 3L, 4L)
```

```
In [35]: np.stack(arrays, axis=1).shape
```

```
Out[35]: (3L, 10L, 4L)
```

```
In [36]: np.stack(arrays, axis=2).shape
```

```
Out[36]: (3L, 4L, 10L)
```

```
In [37]: a = np.array([1,2,3])
b = np.array([4,5,6])
np.stack((a,b))
```

```
Out[37]: array([[1, 2, 3],
 [4, 5, 6]])
```

```
In [38]: np.stack((a,b), axis=-1)
```

```
Out[38]: array([[1, 4],
 [2, 5],
 [3, 6]])
```

2. numpy.hstack()函数

函数原型：numpy.hstack(tup)，其中tup是arrays序列，阵列必须具有相同的形状，除了对应于轴的维度（默认情况下，第一个）。

等价于np.concatenate (tup, axis=1)

示例：

```
In [39]: a = np.array((1,2,3))
b = np.array((4,5,6))
np.hstack((a,b))
```

```
Out[39]: array([1, 2, 3, 4, 5, 6])
```

```
In [41]: a = np.array([[1],[2],[3]])
b = np.array([[4],[5],[6]])
np.hstack((a,b))
```

```
Out[41]: array([[1, 4],
 [2, 5],
 [3, 6]])
```

3. numpy.vstack()函数

函数原型：numpy.vstack(tup)

等价于：np.concatenate(tup, axis=0)

示例：

```
In [42]: a = np.array((1,2,3))
b = np.array((4,5,6))
np.vstack((a,b))
```

```
Out[42]: array([[1, 2, 3],
 [4, 5, 6]])
```

```
In [43]: a = np.array([[1],[2],[3]])
b = np.array([[4],[5],[6]])
np.vstack((a,b))
```

```
Out[43]: array([[1],
 [2],
 [3],
 [4],
 [5],
 [6]])
```

4. numpy.dstack()函数

函数原型：numpy.dstack(tup)

等价于：np.concatenate(tup, axis=2)

示例：

```
In [44]: a = np.array((1,2,3))
b = np.array((4,5,6))
np.dstack((a,b))
```

```
Out[44]: array([[[1, 4],
 [2, 5],
 [3, 6]]])
```

```
In [45]: a = np.array([[1],[2],[3]])
b = np.array([[4],[5],[6]])
np.dstack((a,b))
```

```
Out[45]: array([[[1, 4]],
 [[2, 5]],
 [[3, 6]]])
```

5. numpy.concatenate()函数

函数原型：numpy.concatenate((a1,a2,...),axis=0)

示例：

```
In [46]: a = np.array((1,2,3))
b = np.array((4,5,6))
np.concatenate((a,b))

Out[46]: array([1, 2, 3, 4, 5, 6])

In [47]: np.concatenate((a,b),axis=0)

Out[47]: array([1, 2, 3, 4, 5, 6])

In [49]: a = np.array([[1],[2],[3]])
b = np.array([[4],[5],[6]])
np.concatenate((a,b),axis=1)

Out[49]: array([[1, 4],
                [2, 5],
                [3, 6]])

In [50]: a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)

Out[50]: array([[1, 2],
                [3, 4],
                [5, 6]])

In [51]: np.concatenate((a, b.T), axis=1)

Out[51]: array([[1, 2, 5],
                [3, 4, 6]])
```

6. numpy.vsplit()函数

函数原型: numpy.vsplit(ary,indices_or_sections)

示例:

```
In [52]: a = np.arange(16.0).reshape(4,4)
a

Out[52]: array([[ 0.,  1.,  2.,  3.],
   [ 4.,  5.,  6.,  7.],
   [ 8.,  9., 10., 11.],
   [12., 13., 14., 15.]])
```

```
In [53]: np.split(a,2)

Out[53]: [array([[ 0.,  1.,  2.,  3.],
   [ 4.,  5.,  6.,  7.]]), array([[ 8.,  9., 10., 11.],
   [12., 13., 14., 15.]])]
```

```
In [54]: b = np.arange(8).reshape(2,2,2)
b

Out[54]: array([[[0, 1],
   [2, 3]],

   [[4, 5],
   [6, 7]]])
```

```
In [56]: np.vsplit(b,2)

Out[56]: [array([[[0, 1],
   [2, 3]]]), array([[[4, 5],
   [6, 7]]])]
```

Array deleting

```
#删除一列

>>> dataset=[[1,2,3],[2,3,4],[4,5,6]]
>>> import numpy as np
>>> dataset = np.delete(dataset, -1, axis=1)
>>> dataset
array([[1, 2],
   [2, 3],
   [4, 5]])
```

```
# 删除多列
arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
np.delete(arr, [1,2], axis=1)
array([[ 1,  4],
   [ 5,  8],
   [ 9, 12]])
```

Array new axis

```
x = np.random.randint(1, 8, size=5)

x
Out[48]: array([4, 6, 6, 6, 5])

x1 = x[np.newaxis, :]

x1
Out[50]: array([[4, 6, 6, 6, 5]])

x2 = x[:, np.newaxis]

x2
Out[52]:
array([[4,
       6,
       6,
       6,
       5]])
```

由以上代码可以看出，当把newaxis放在前面的时候

以前的shape是5，现在变成了 1×5 ，也就是前面的维数发生了变化，后面的维数发生了变化

而把newaxis放后面的时候，输出的新数组的shape就是 5×1 ，也就是后面增加了一个维数

所以，newaxis放在第几个位置，就会在shape里面看到相应的位置增加了一个维数

```
In [59]: x = np.random.randint(1, 8, size=(2, 3, 4))
```

```
In [60]: y = x[:, np.newaxis, :, :]
```

```
In [61]: z = x[:, :, np.newaxis, :]
```

```
In [62]: x.shape
```

```
Out[62]: (2, 3, 4)
```

```
In [63]: y.shape
```

```
Out[63]: (2, 1, 3, 4)
```

```
In [64]: z.shape
```

```
Out[64]: (2, 3, 1, 4)
```

<http://blog.csdn.net/xtingjie>

Datatypes

Every numpy array is a grid of elements of the **same type**. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
import numpy as np

x = np.array([1, 2])      # Let numpy choose the datatype
print(x.dtype)            # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)            # Prints "float64"

x = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
print(x.dtype)              # Prints "int64"
```

You can read all about numpy datatypes [in the documentation](#).

Array math

Basic mathematical functions operate **elementwise** on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2          0.33333333]
#  [ 0.42857143  0.5         ]]
print(x / y)
print(np.divide(x, y))
```

```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
# [ 1.73205081  2.          ]]
print(np.sqrt(x))
```

Note that unlike MATLAB, `*` is elementwise multiplication, not matrix multiplication. We instead use the `dot` function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. `dot` is available both as a function in the numpy module and as an instance method of array objects:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; --the result is array (one dimensional array showed in row by default)
#both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
# [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x)) # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0)) # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1)) # Compute sum of each row; prints "[3 7]"
```

You can find the full list of mathematical functions provided by numpy [in the documentation](#).

Apart from computing mathematical functions using arrays, we frequently need to `reshape` or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the `T` attribute of an array object:

```

import numpy as np

x = np.array([[1,2], [3,4]])
print(x)      # Prints "[[1 2]
#                 [3 4]]"
print(x.T)   # Prints "[[1 3]
#                 [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)      # Prints "[1 2 3]"
print(v.T)   # Prints "[1 2 3]"

```

Numpy provides many more functions for manipulating arrays; you can see the full list [in the documentation](#).

取整计算

around:

- `np.around` 返回四舍五入后的值，可指定精度。

```

around(a, decimals=0, out=None)

# a 输入数组
# decimals 要舍入的小数位数。 默认值为0。 如果为负，整数将四舍五入到小数点左侧的位置

```

floor (向下取整)

- `np.floor` 返回不大于输入参数的最大整数。即对于输入值 x ，将返回最大的整数 i ，使得 $i \leq x$ 。注意在 Python 中，向下取整总是从 0 舍入。

ceil: (向上取整)

- `np.ceil` 函数返回输入值的上限，即对于输入 x ，返回最小的整数 i ，使得 $i \geq x$ 。

np.where

- `numpy.where(condition, x, y)` 根据 condition 从 x 和 y 中选择元素，当为 True 时，选 x ，否则选 y 。

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html>

```

import numpy as np

data = np.random.random([2, 3])
print data
...
[[ 0.93122679  0.82384876  0.28730977]
 [ 0.43006042  0.73168913  0.02775572]]
...

result = np.where(data > 0.5, data, 0)

```

```
print result
...
[[ 0.93122679  0.82384876  0.          ]
 [ 0.          0.73168913  0.          ]
 ...]
```

np.any(), np.all()

any(iterables)和all(iterables)对于检查两个对象相等时非常实用，但是要注意，any和all是python内置函数，同时numpy也有自己实现的any和all，功能与python内置的一样，只不过把numpy.ndarray类型加进去了。因为python内置的对高于1维的ndarray没法理解，所以numpy基于的计算最好用numpy自己实现的any和all。

本质上讲，any()实现了或(OR)运算，而all()实现了与(AND)运算。

- 对于any(iterables)，如果可迭代对象iterables中任意存在每一个元素为True则返回True。特例：若可迭代对象为空，比如空列表[]，则返回False。
- 对于all(iterables)，如果可迭代对象iterables中所有元素都为True则返回True。特例：若可迭代对象为空，比如空列表[]，则返回True。

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and **we want to use the smaller array multiple times to perform some operation on the larger array.**

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x) # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector `v` to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this

approach like this:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1)) # Stack 4 copies of v on top of each other
print(vv) # Prints "[[1 0 1]
           #          [1 0 1]
           #          [1 0 1]
           #          [1 0 1]]"
y = x + vv # Add x and vv elementwise
print(y) # Prints "[[ 2  2  4
           #          [ 5  5  7]
           #          [ 8  8 10]
           #          [11 11 13]]"
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of `v`. Consider this version, using broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # Add v to each row of x using broadcasting
print(y) # Prints "[[ 2  2  4]
           #          [ 5  5  7]
           #          [ 8  8 10]
           #          [11 11 13]]"
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if `v` actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise **maximum of shapes** of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation [from the documentation](#) or [this explanation](#).

Functions that support broadcasting are known as *universal functions*. You can find the list of all universal functions [in the documentation](#).

Here are some applications of broadcasting:

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3]) # v has shape (3,)
w = np.array([4,5]) # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

Reshape & Resize

NumPy 中有两个跟形状转换相关的函数（及方法）`reshape` 以及 `resize`，它们都能方便的改变矩阵的形状，但是它们之间又有一个显著的差别，我们会着重的来讲。

`numpy.reshape()`:

我们先来看看会在各种数据计算中经常用到的改变数组形状的函数 `reshape()`。

```
import numpy as np

arrayA = np.arange(8)
# arrayA = array([0, 1, 2, 3, 4, 5, 6, 7])

np.reshape(arrayA, (2, 4))
#array([[0, 1, 2, 3],
#       [4, 5, 6, 7]])
```

这里它把一个有 8 个元素的向量，转换成了形状为 `(4, 2)` 的一个矩阵。因为转换前后的元素数目一样，所以能够成功的进行转换，假如前后数目不一样的话，就会有错误 `ValueError` 报出。

```
In [1]: np.reshape(arrayA, (3, 4))
-----
ValueError                                Traceback (most recent call last)
ValueError: cannot reshape array of size 8 into shape (3,4)
```

我们仔细看转换后的数据，第一行是 `arrayA` 的前四个数据，第二行是 `arrayA` 的后四个数据，也就是它是按行来填充数据的，有些时候我们需要把数据填充的顺序改成按列来填充，那我们需要改变函数中的另外一个输入参数 `order=`

```
In [1]: np.reshape(arrayA, (2, 4), order='F')
Out[1]: array([[0, 2, 4, 6],
               [1, 3, 5, 7]])
```

`order` 默认的参数是 `C`，也就是按行填充，当参数变为 `F` 时，就变成按列填充。

说明

这里用按行或者按列来填充，是为了便于理解，其实具体的不同是由于函数是按照类似 `C` 的索引顺序还是按照类似 `Fortran` 的索引顺序来读取输入矩阵的内容，具体可以参照[Numpy reshape 的官方说明文档](#)。

`reshape()` 函数/方法内存:

`reshape` 函数或者方法生成的新数组和原始数组是共用一个内存的，有点类似于 Python 里面的 `shallow copy`，当你改变一个数组的元素，另外一个数组的元素也相应的改变了。

```
In [1]: arrayA = np.arange(8)
        arrayB = arrayA.reshape((2, 4))
        arrayB
Out[2]: array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
In [2]: arrayA[0] = 10
        arrayA
Out[2]: array([10, 1, 2, 3, 4, 5, 6, 7])
In [3]: arrayB
Out[3]: array([[10, 1, 2, 3],
               [4, 5, 6, 7]])
```

numpy.resize():

`numpy.resize()` 跟 `reshape` 类似，可以改变矩阵的形状，但它有几点不同，

1. 没有 `order` 参数了，它只有跟 `reshape` 里面 `order='C'` 的方式。
2. 假如要转换成的矩阵形状中的元素数量跟原矩阵不同，它会强制进行转换，而不报错。

我们具体来看一下第二点

```
In [1]: arrayA = np.arange(8)
        arrayB = np.resize(arrayA, (2, 4))
Out[1]: array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
```

这是尺寸大小正常情况，跟 `reshape` 的结果是一样的。

```
In [1]: arrayC = np.resize(arrayA, (3, 4))
        arrayC
Out[1]: array([[0, 1, 2, 3],
               [4, 5, 6, 7],
               [0, 1, 2, 3]])
In [2]: arrayD = np.resize(arrayA, (4, 4))
        arrayD
Out[2]: array([[0, 1, 2, 3],
               [4, 5, 6, 7],
               [0, 1, 2, 3],
               [4, 5, 6, 7]])
```

当新形状行数超出的话，它会开始重复填充原始矩阵的内容，实现形状大小的自动调整而不报错。

```
In [1]: arrayE = np.resize(arrayA, (2, 2))
        arrayE
Out[1]: array([[0, 1],
               [2, 3]])
In [2]: np.resize(arrayA, (1, 4))
Out[2]: array([[0, 1, 2, 3]])
```

当新形状比原形状所需要的数据小的时候，它会从原矩阵读取所需数据，然后按照先按行填充的方式对新矩阵元素赋值。

注意

当新矩阵的形状在原矩阵形状内时，比如 `(2, 2)` 和 `(2, 4)` 的情形，新矩阵的元素不是按照子集来获取的。比如上例中，`np.resize(arrayA, (2, 2))` 不是 `array([[0, 1], [4, 5]])`。假如你需要这样的截取方法，我们会在后续的索引和切片操作中介绍到的。

```
In [1]: np.resize(arrayA, (3, 5))
Out[1]: array([[0, 1, 2, 3, 4],
               [5, 6, 7, 0, 1],
               [2, 3, 4, 5, 6]])
```

当新形状比原形状要大时，它会先按行去填充旧矩阵的元素，并且在元素被用光后，再重复的填充这些元素，直到新矩阵的最后一元素。

resize 函数/方法内存：

跟 `reshape` 不一样的是，`resize` 函数/方法生成的新数组跟原数组并不共用一个内存，所以彼此元素的改变不会影响到对方。

```
In [1]: arrayA = np.arange(8)
        arrayB = arrayA.reshape((2, 4))
        arrayB
Out[2]: array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
In [2]: arrayA[0] = 10
        arrayA
Out[2]: array([10, 1, 2, 3, 4, 5, 6, 7])
In [3]: arrayB
Out[3]: array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
```

Repet & Tile

numpy数组扩展函数有repeat和tile，由于数组不能进行动态扩展，故函数调用之后都重新分配新的空间来存储扩展后的数据。

repeat函数：

功能：对数组中的 元素 进行连续重复复制

用法：

- `numpy.repeat(a, repeats, axis=None)`
- `a.repeats(repeats, axis=None)`

其中a为数组，repeats为重复的次数，axis表示数组维度

```
>>> import numpy as np
>>> a = np.arange(16).reshape(4,4)
>>> a
array([[ 0,  1,  2,  3],
```

```

[ 4,  5,  6,  7],
[ 8,  9, 10, 11],
[12, 13, 14, 15]])
>>> b = a.repeat(3)
# 未指定axis 参数时，会默认把高维数组flatten至一维
>>> b
array([ 0,  0,  0,  1,  1,  1,  2,  2,  2,  3,  3,  3,  4,  4,  4,  5,
       5,  6,  6,  6,  7,  7,  7,  8,  8,  8,  9,  9,  9, 10, 10, 10, 10,
      11, 11, 12, 12, 12, 13, 13, 13, 14, 14, 14, 14, 15, 15, 15])
>>> b.reshape(4,12)
array([[ 0,  0,  0,  1,  1,  1,  2,  2,  2,  3,  3,  3],
       [ 4,  4,  4,  5,  5,  5,  6,  6,  6,  7,  7,  7],
       [ 8,  8,  8,  9,  9,  9, 10, 10, 10, 11, 11, 11],
      [12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15]])
>>> c
array([[ 0,  0,  0,  1,  1,  1,  2,  2,  2,  3,  3,  3],
       [ 4,  4,  4,  5,  5,  5,  6,  6,  6,  7,  7,  7],
       [ 8,  8,  8,  9,  9,  9, 10, 10, 10, 11, 11, 11],
      [12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15]])
>>> c = c.repeat(3, axis=0)
>>> c
array([[ 0,  0,  0,  1,  1,  1,  2,  2,  2,  3,  3,  3],
       [ 0,  0,  0,  1,  1,  1,  2,  2,  2,  3,  3,  3],
       [ 0,  0,  0,  1,  1,  1,  2,  2,  2,  3,  3,  3],
       [ 4,  4,  4,  5,  5,  5,  6,  6,  6,  7,  7,  7],
       [ 4,  4,  4,  5,  5,  5,  6,  6,  6,  7,  7,  7],
       [ 4,  4,  4,  5,  5,  5,  6,  6,  6,  7,  7,  7],
       [ 4,  4,  4,  5,  5,  5,  6,  6,  6,  7,  7,  7],
       [ 8,  8,  8,  9,  9,  9, 10, 10, 10, 11, 11, 11],
       [ 8,  8,  8,  9,  9,  9, 10, 10, 10, 11, 11, 11],
       [ 8,  8,  8,  9,  9,  9, 10, 10, 10, 11, 11, 11],
      [12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15],
      [12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15],
      [12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15]])

```

tile函数:

功能：对整个数组进行复制拼接

用法：numpy.tile(a, reps)

其中a为数组，reps为重复的次数

```

>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
      [12, 13, 14, 15]])
>>> b = np.tile(a, 2)
>>> b
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11],
      [12, 13, 14, 15, 12, 13, 14, 15]])

```

```

[ 8,  9, 10, 11,  8,  9, 10, 11],
[12, 13, 14, 15, 12, 13, 14, 15]]))

>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

>>> c = np.tile(a, (2,2))
>>> c
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11],
       [12, 13, 14, 15, 12, 13, 14, 15],
       [ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11],
       [12, 13, 14, 15, 12, 13, 14, 15]])

```

array to list

```

>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]

# list to array
a = np.array(a.tolist()).

```

sort

sort

-

argsort

`numpy.argsort(a, axis=-1, kind='quicksort', order=None)`

Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the `kind` keyword. It returns an array of indices of the same shape as `a` that index data along the given axis in sorted order.

Parameters:

- `a` : array_like. Array to sort.

- **axis** : int or None, optional. Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.
- **kind** : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optionalSorting algorithm.
- **order** : str or list of str, optional When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

Returns:

- **index_array** : ndarray, int. Array of indices that sort a along the specified axis. If a is one-dimensional, $a[\text{index_array}]$ yields a sorted a . More generally, `np.take_along_axis(a, index_array, axis=a)` always yields the sorted a , irrespective of dimensionality.

Examples

```
# One dimensional array:
>>> x = np.array([3, 1, 2])
>>> np.argsort(x)
array([1, 2, 0])

# Two-dimensional array:
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])
>>> np.argsort(x, axis=0) # sorts along first axis (down)
array([[0, 1],
       [1, 0]])

>>> np.argsort(x, axis=1) # sorts along last axis (across)
array([[0, 1],
       [0, 1]])

# Indices of the sorted elements of a N-dimensional array:

>>> ind = np.unravel_index(np.argsort(x, axis=None), x.shape)
>>> ind
(array([0, 1, 1, 0]), array([0, 0, 1, 1]))
>>> x[ind] # same as np.sort(x, axis=None)
array([0, 2, 2, 3])
```

Sorting with keys:
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
 dtype=[('x', '<i4'), ('y', '<i4')])
>>> np.argsort(x, order=('x', 'y'))
array([1, 0])
>>> np.argsort(x, order=('y', 'x'))
array([0, 1])
```

See also

- [sort](#)

Describes sorting algorithms used.

- [lexsort](#)

Indirect stable sort with multiple keys.

- [ndarray.sort](#)

Inplace sort.

- [argpartition](#)

Indirect partial sort.

## Numpy Documentation

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the [numpy reference](#) to find out much more about numpy.

# 数据读写

## csv 文件

### 写入

```
import csv

#python2要用file替代open
with open("test.csv","w") as csvfile:
 writer = csv.writer(csvfile)

 #先写入columns_name
 writer.writerow(["index","a_name","b_name"])
 #写入多行用writerows
 writer.writerows([[0,1,3],[1,2,3],[2,3,4]])
```

### 读取

```
import csv
with open("test.csv","r") as csvfile:
 reader = csv.reader(csvfile)
 #这里不需要readlines
 for line in reader:
 print line
```

**注意：**采用以上方法，写入的都是字符串类型的！！！当是单个数值时读取时采用float(line[0]) 强制转化一下即可，但如果写入了列表，就无法转换了！！！

## Save Numpy array to csv file

- Use "savetxt" method of numpy to save a numpy array to csv file
- Use "genfromtxt" method to read a csv file into a numpy array

```
Imports
import numpy as np

Let's creat a numpy array
nparray = np.array([[1, 2, 4], [1, 3, 9], [1, 4, 16]])

Saving the array
np.savetxt("firstarray.csv", nparray, delimiter=",")

Reading the csv into an array
firstarray = np.genfromtxt("firstarray.csv", delimiter=",")

print(firstarray)
```

```
[[1. 2. 4.]
 [1. 3. 9.]
 [1. 4. 16.]]
```

## text 文件

打开文件：

```
with open('/path/to/file', 'r') as f:
 data = f.read()
```

`open` 函数的常用模式主要有：

|     |                    |
|-----|--------------------|
| 'r' | 读模式                |
| 'w' | 写模式                |
| 'a' | 追加模式               |
| 'b' | 二进制模式（可添加到其他模式中使用） |
| '+' | 读/写模式（可添加到其他模式中使用） |

读取文件：

通常而言，读取文件有以下几种方式：

- 一次性读取所有内容，使用 `read()` 或 `readlines()`；
- 按字节读取，使用 `read(size)`；
- 按行读取，使用 `readline()`；

`readlines()` 方法会把文件读入一个字符串列表，文件中的每一行在列表中存为一个字符串。

假设有一个文件 `data.txt`，它的文件内容如下（数字之间的间隔符是'\t'）：

```
10 1 9 9
6 3 2 8
20 10 3 23
1 4 1 10
10 8 6 3
10 2 1 6
```

我们使用 `readlines()` 将文件读入一个字符串列表：

```
with open('data.txt', 'r') as f:
 lines = f.readlines()
 line_num = len(lines)
 print lines
 print line_num
```

执行结果：

```
['10\t1\t9\t9\n', '6\t3\t2\t8\n', '20\t10\t3\t23\n', '1\t4\t1\t10\n', '10\t8\t6\t3\n',
'10\t2\t1\t6']
```

可以看到，列表中的每个元素都是一个字符串，刚好对应文件中的每一行。

### 按字节读取：

如果文件较小，一次性读取所有内容确实比较方便。但是，如果文件很大，比如有 100G，那就不能一次性读取所有内容了。这时，我们构造一个固定长度的缓冲区，来不断读取文件内容。

看看例子：

```
with open('path/to/file', 'r') as f:
 while True:
 piece = f.read(1024) # 每次读取 1024 个字节（即 1 KB）的内容
 if not piece:
 break
 print piece
```

在上面，我们使用 `f.read(1024)` 来每次读取 1024 个字节（1KB）的文件内容，将其存到 `piece`，再对 `piece` 进行处理。

事实上，我们还可以结合 `yield` 来使用：

```
def read_in_chunks(file_object, chunk_size=1024):
 """Lazy function (generator) to read a file piece by piece.
 Default chunk size: 1k."""
 while True:
 data = file_object.read(chunk_size)
 if not data:
 break
 yield data

with open('path/to/file', 'r') as f:
 for piece in read_in_chunks(f):
 print piece
```

## 逐行读取

在某些情况下，我们希望逐行读取文件，这时可以使用 `readline()` 方法。

看看例子：

```
with open('data.txt', 'r') as f:
 while True:
 line = f.readline() # 逐行读取
 if not line:
 break
 print line, # 这里加了 ',' 是为了避免 print 自动换行
```

执行结果：

```
10 1 9 9
6 3 2 8
20 10 3 23
1 4 1 10
10 8 6 3
10 2 1 6
```

## 文件迭代器

在 Python 中，**文件对象是可迭代的**，这意味着我们可以直接在 `for` 循环中使用它们，而且是逐行迭代的，也就是说，效果和 `readline()` 是一样的，而且更简洁。

看看例子：

```
with open('data.txt', 'r') as f:
 for line in f:
 print line,
```

在上面的代码中，`f` 就是一个文件迭代器，因此我们可以直接使用 `for line in f`，它是逐行迭代的。

看看执行结果：

```
10 1 9 9
6 3 2 8
20 10 3 23
1 4 1 10
10 8 6 3
10 2 1 6
```

再看一个例子：

```
with open(file_path, 'r') as f:
 lines = list(f)
 print lines
```

执行结果：

```
['10\t1\t9\t9\n', '6\t3\t2\t8\n', '20\t10\t3\t23\n', '1\t4\t1\t10\n', '10\t8\t6\t3\n',
'10\t2\t1\t6']
```

可以看到，我们可以对文件迭代器执行和普通迭代器相同的操作，比如上面使用 `list(open(filename))` 将 `f` 转为一个字符串列表，这样所达到的效果和使用 `readlines` 是一样的。

## 写文件

写文件使用 `write` 方法，如下：

```
with open('/Users/ethan/data2.txt', 'w') as f:
 f.write('one\n')
 f.write('two')
```

- 如果上述文件已存在，则会清空原内容并覆盖掉；
- 如果上述路径是正确的（比如存在 /Users/ethan 的路径），但是文件不存在（data2.txt 不存在），则会新建一个文件，并写入上述内容；
- 如果上述路径是不正确的（比如将路径写成 /Users/eth ），这时会抛出 IOError；

如果我们想往已存在的文件追加内容，可以使用 'a' 模式，如下：

```
from datetime import datetime
date = datetime.now().strftime('%Y_%m_%d')

output_path = '/home/ubuntu16/catkin_ws/src/sonar_navigation/h5/'
output_file = output_path + 'MLP_training_' + date + '.txt'
with open(output_file, 'a') as f:
 for i in range(3):
 f.write(output_file+ '\n')
```

## SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. [SciPy](#) builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications.

The best way to get familiar with SciPy is to [browse the documentation](#). We will highlight some parts of SciPy that you might find useful for this class.

You can install packages via commands such as:

```
python -m pip install --user numpy scipy matplotlib ipython jupyter pandas sympy nose pillow
```

We recommend using an *user* install, using the `--user` flag to pip (note: do not use `sudo pip`, which can cause problems). This installs packages for your local user, and does not write to the system directories.

## Image operations

SciPy provides some basic functions to work with images.

For example, it has functions to read images from disk into numpy arrays, to write numpy arrays to disk as images, and to resize images. Here is a simple example that showcases these functions:

```
scipy.misc.imsave(name, arr)[source]
```

Save an array as an image.

```
Parameters :
name : str

Output filename.

arr : ndarray, MxN or MxNx3 or MxNx4 Array containing image values. If the shape is MxN, the array represents a grey-level image. Shape MxNx3 stores the red, green and blue bands along the last dimension. An alpha layer may be included, specified as the last colour band of an MxNx4 array.
```

### Examples

Construct an array of gradient intensity values and save to file:

```
>>> x = np.zeros((255, 255))
>>> x = np.zeros((255, 255), dtype=np.uint8)
>>> x[:, :] = np.arange(255) # all the rows of X is [0,1,2,...,254]
>>> imsave('/tmp/gradient.png', x)
```

Construct an array with three colour bands (R, G, B) and store to file:

```
>>> rgb = np.zeros((255, 255, 3), dtype=np.uint8)
>>> rgb[:, :, 0] = np.arange(255)
>>> rgb[:, :, 1] = 55
>>> rgb[:, :, 2] = 1 - np.arange(255)
>>> imsave('/tmp/rgb_gradient.png', rgb)
```

```

from scipy.misc import imread, imsave, imresize

Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape) # Prints "uint8 (400, 248, 3)"

We can tint the image by scaling each of the color channels
by a different scalar constant. The image has shape (400, 248, 3);
we multiply it by the array [1, 0.95, 0.9] of shape (3,);
numpy broadcasting means that this leaves the red channel unchanged,
and multiplies the green and blue channels by 0.95 and 0.9
respectively.
img_tinted = img * [1, 0.95, 0.9]

Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)

```



Left: The original image. Right: The tinted and resized image.

## MATLAB files

The functions `scipy.io.loadmat` and `scipy.io.savemat` allow you to **read and write MATLAB files**. You can read about them [in the documentation](#).

## Distance between points

SciPy defines some useful functions for computing distances between sets of points.

The function `scipy.spatial.distance.pdist` computes the distance between all pairs of points in a given set:

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

Create the following array where each row is a point in 2D space:
[[0 1]
[1 0]
[2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

Compute the Euclidean distance between all rows of x.
d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
and d is the following array:
[[0. 1.41421356 2.23606798]
[1.41421356 0. 1.]
[2.23606798 1. 0.]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

You can read all the details about this function  
[in the documentation](#).

A similar function (`scipy.spatial.distance.cdist`) computes the distance between all pairs across two sets of points; you can read about it  
[in the documentation](#).

## Matplotlib

[Matplotlib](#) is a plotting library.

In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

## plot

The most important function in matplotlib is `plot`,  
which allows you to plot 2D data. Here is a simple example:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
在jupyter notebook 中画图需要加此句，否则第一次运行不会显示图像

创建函数
在[-3,3]区间产生50个点
x = np.linspace(-3,3,50)
y1 = 2*x + 1
```

```
y2 = x**2

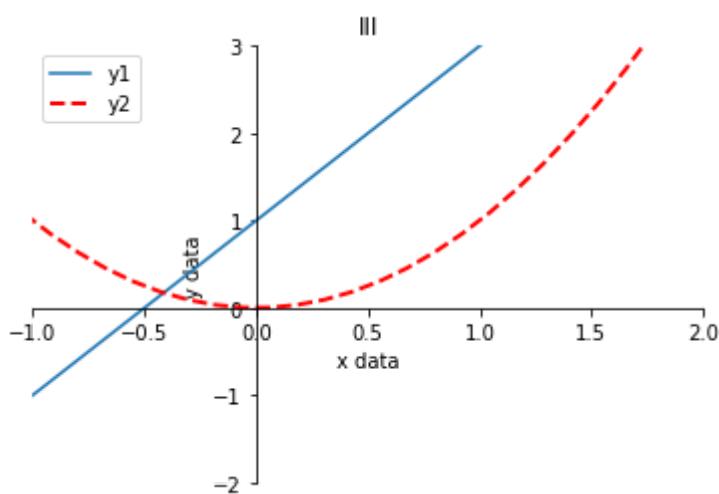
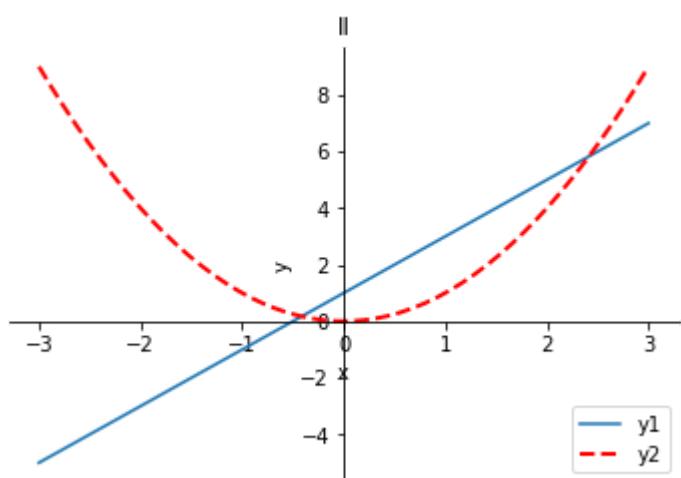
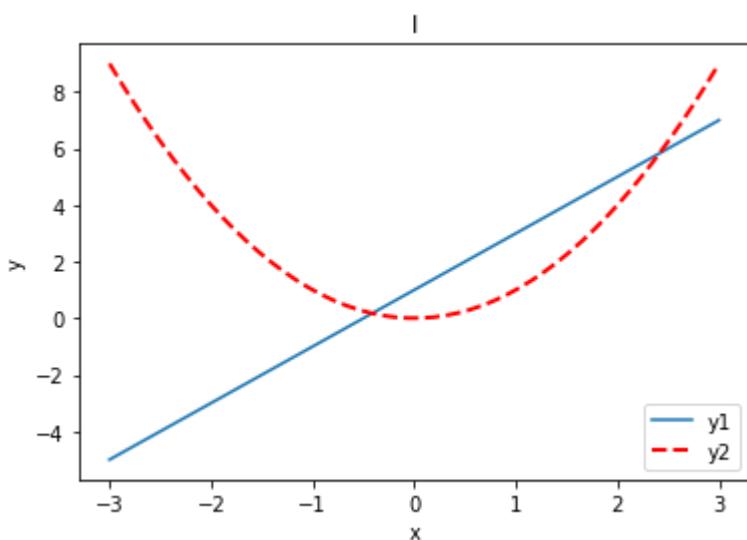
绘图-1
plt.figure()
plt.plot(x,y1, label="y1")
plt.plot(x,y2,color='red', linewidth=2, linestyle='--', label='y2')
#添加相关图例
plt.xlabel("x")
plt.ylabel('y')
plt.legend()
plt.title('I')

调整-2
#gca 'get current axes' 获取图像的四个轴 right、top、bottom、left
ax = plt.gca()
设置两个轴颜色为空隐藏两个轴
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
移动其余两个轴到指定位置; 设置原点为(0,0)
ax.spines['bottom'].set_position((0,0))
ax.spines['left'].set_position((0,0))
plt.title('II')

调整-3
利用axes对象设置轴线的显示范围, 与plt.xlim(-1,2)和plt.ylim(-2,3)的作用相同
ax.set_xlim(-1,2)
ax.set_ylim(-2,3)
利用axes对象设置坐标轴的标签
ax.set_xlabel('x data')
ax.set_ylabel('y data')
设置坐标轴上的数字显示的位置, top:显示在顶部 bottom:显示在底部, 默认是none
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.set_title('III')

plt.show()
现在坐标轴会显示默认数据范围
```

Running this code produces the following plot:



You can read much more about the `plot` function [in the documentation](#).

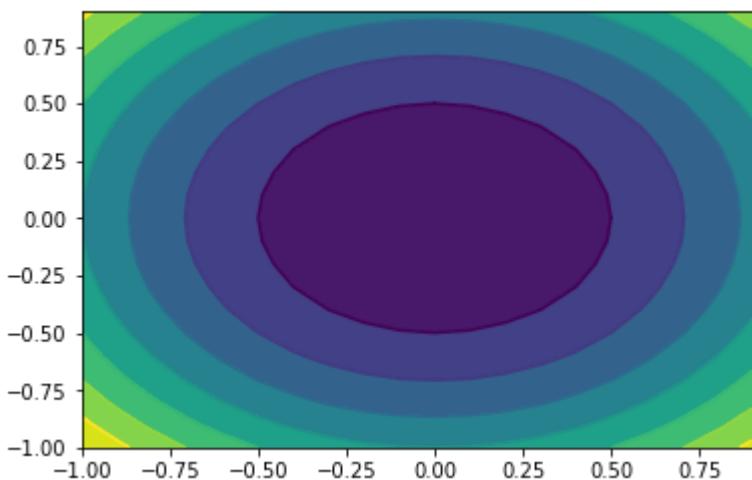
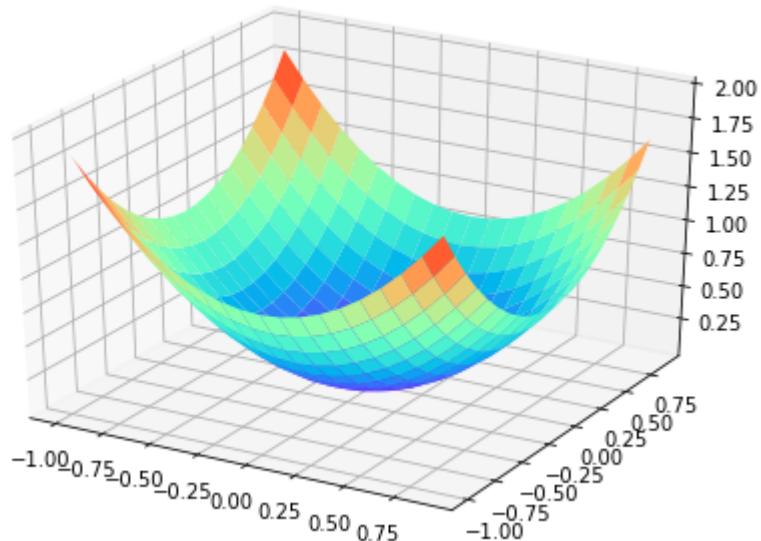
```
-*- coding: utf-8 -*-
...
save维图与等高线图
更多用法参见:
https://blog.csdn.net/Mr_Cat123/article/details/80677525
...
...
```

```
from matplotlib import pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

x1 = np.arange(-1, 1, 0.1)
x2 = np.arange(-1, 1, 0.1)
通过复制将x1,x2变成2维数据，便于点对点操作
x1_gride, x2_gride = np.meshgrid(x1, x2)
y = x1_gride**2 + x2_gride**2

fig = plt.figure()
ax = Axes3D(fig)
具体函数方法可用 help(function) 查看，如: help(ax.plot_surface)
ax.plot_surface(x1_gride, x2_gride, y, rstride=1, cstride=1, cmap='rainbow')
plt.show()

fig = plt.figure()
#填充颜色，f即filled
plt.contourf(x1,x2,y)
#画等高线
plt.contour(x1,x2,y)
plt.show()
```



```
...
绘制三维散点图
...

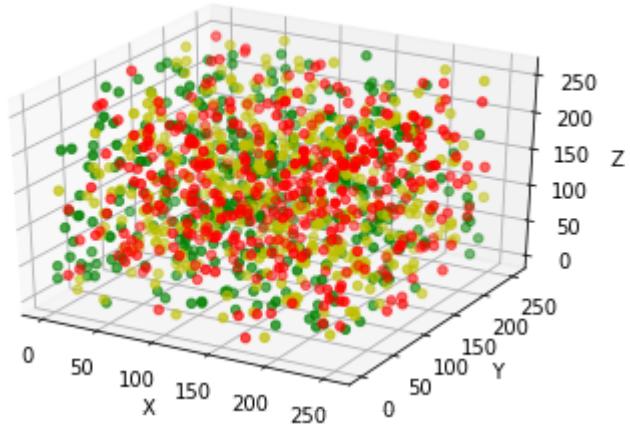
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

data = np.random.randint(0, 255, size=[40, 40, 40])

x, y, z = data[0], data[1], data[2]
创建一个三维的绘图工程
ax = plt.subplot(111, projection='3d')
将数据点分成三部分画，在颜色上有区分度
ax.scatter(x[:10], y[:10], z[:10], c='y') # 绘制数据点
ax.scatter(x[10:20], y[10:20], z[10:20], c='r')
ax.scatter(x[30:40], y[30:40], z[30:40], c='g')

ax.set_zlabel('Z') # 坐标轴
ax.set_ylabel('Y')
```

```
ax.set_xlabel('X')
plt.show()
```



## Subplot

You can plot different things in the same figure using the `subplot` function.

Here is an example:

```
import numpy as np
import matplotlib.pyplot as plt

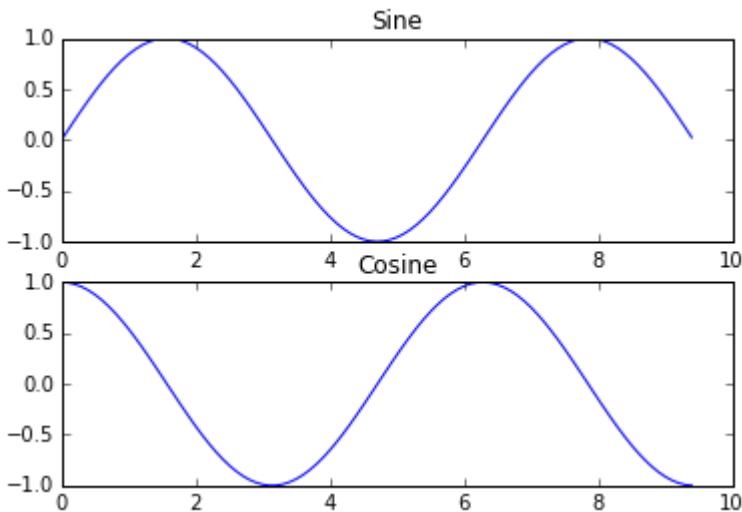
Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

Set up a subplot grid that has height 2 and width 1,
and set the first such subplot as active.
plt.subplot(2, 1, 1)

Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

Show the figure.
plt.show()
```



You can read much more about the `subplot` function  
[in the documentation](#).

## subplots

```
matplotlib.pyplot.subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True,
subplot_kw=None, gridspec_kw=None, **fig_kw)
```

**作用：**

创建一个画像(figure)和一组子图(subplots)

**参数：**

- `nrows,ncols`: 整型，可选参数，默认为1，表示子图网格(grid)的行数与列数
- `sharex,sharey`: 控制x(`sharex`)或y(`sharey`)轴之间的属性共享，布尔值或者{'none','all','row','col'}，默认：`False`
  - `True`或者`'all'`: x或y轴属性将在所有子图(subplots)中共享.
  - `False`或`'none'`: 每个子图的x或y轴都是独立的部分
  - `'row'`: 每个子图在一个x或y轴共享行(row)
  - `'col'`: 每个子图在一个x或y轴共享列(column)
  - 当子图在x轴有一个共享列时('col'),只有底部子图的x tick标记是可视的。
  - 同理，当子图在y轴有一个共享行时('row'),只有第一列子图的y tick标记是可视的。
- `squeeze`: 布尔类型，可选参数，默认：`True`
  - 如果是`True`，额外的维度从返回的Axes(轴)对象中挤出
    - 如果只有一个子图被构建(`nrows=ncols=1`)，结果是单个Axes对象作为标量被返回
    - 对于N1或1N个子图，返回一个1维数组
    - 对于N\*M, N>1和M>1返回一个2维数组
  - 如果是`False`，不进行挤压操作：返回一个元素为Axes实例的2维数组，即使它最终是1x1
- `subplot_kw`: 字典类型，可选参数。把字典的关键字传递给`add_subplot()`来创建每个子图。
- `gridspec_kw`: 字典类型，可选参数。把字典的关键字传递给`GridSpec`构造函数创建子图放在网格里(grid)。
- `fig_kw`: 把所有详细的关键字参数传给`figure()`函数

返回：

- fig: matplotlib.figure.Figure对象
- ax: Axes(轴)对象或Axes(轴)对象数组。

```
matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, clear=False, **kwargs)
```

作用：创建一个新的画布(Figure)。

参数：

- num: 整型或者字符串，可选参数，默认：None。  
如果不提供该参数，一个新的画布(Figure)将被创建而且画布数量将会增加。  
如果提供该参数，带有id的画布是已经存在的，激活该画布并返回该画布的引用。  
如果这个画布不存在，创建并返回画布实例。  
如果num是字符串，窗口标题将被设置为该图的数字。
- figsize: 整型元组，可选参数，默认：None。  
每英寸的宽度和高度。如果不提供，默认值是figure.figsize。
- dpi: 整型，可选参数，默认：None。每英寸像素点。如果不提供，默认是figure.dpi。
- facecolor: 背景色。如果不提供，默认值：figure.facecolor。
- edgecolor: 边界颜色。如果不提供，默认值：figure.edgecolor。
- framemon: 布尔类型，可选参数，默认值：True。如果是False，禁止绘制画图框。
- FigureClass: 源于matplotlib.figure.Figure的类。（可选）使用自定义图实例。
- clear: 布尔类型，可选参数，默认值：False。如果为True和figure已经存在时，这是清理掉改图。

返回：

- figure: Figure。返回的Figure实例也将被传递给后端的new\_figure\_manager，这允许将自定义的图类挂接到pylab接口中。附加的kwarg将被传递给图形init函数

```
#coding=utf8
import numpy as np
import matplotlib.pyplot as plt
#创建一个数组0-100，数据间隔是0.1
x=np.arange(0,100,0.1)

y=x**2

#调用subplots函数
#指定图像分辨率、大小和w,h比例
#创建一个800*600像素、100dpi(每英寸100点)分辨率的图形
#返回一个画布对象和一个轴数组
fig,axe=plt.subplots(figsize=(4,3),dpi=100)

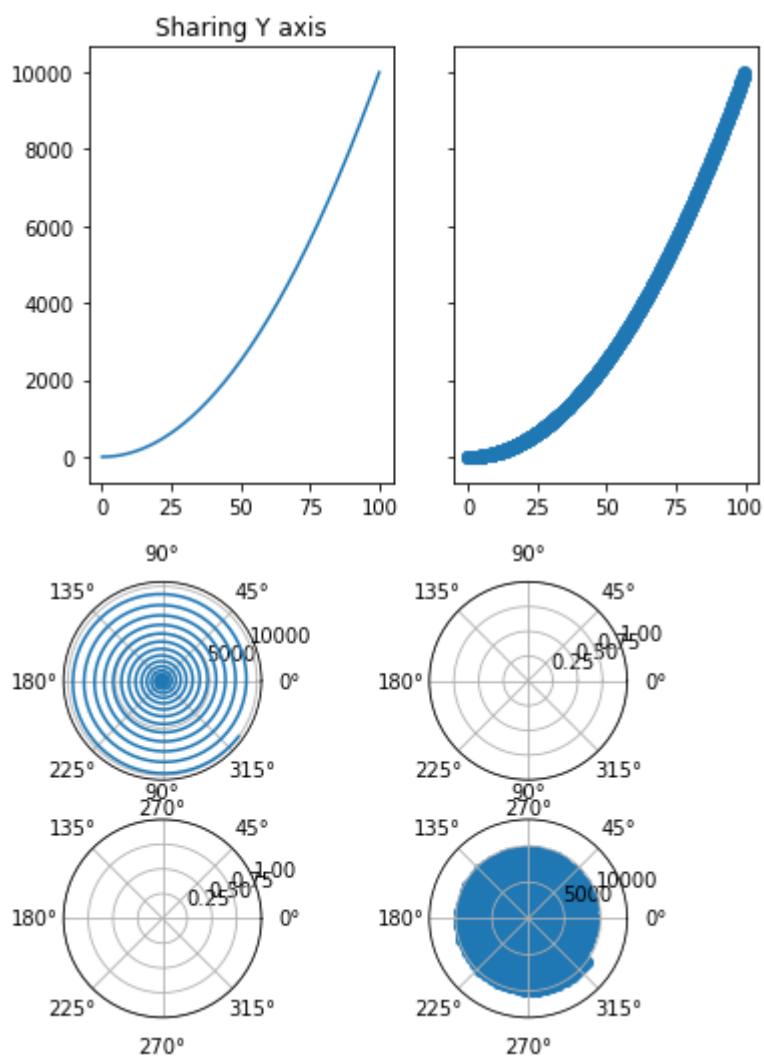
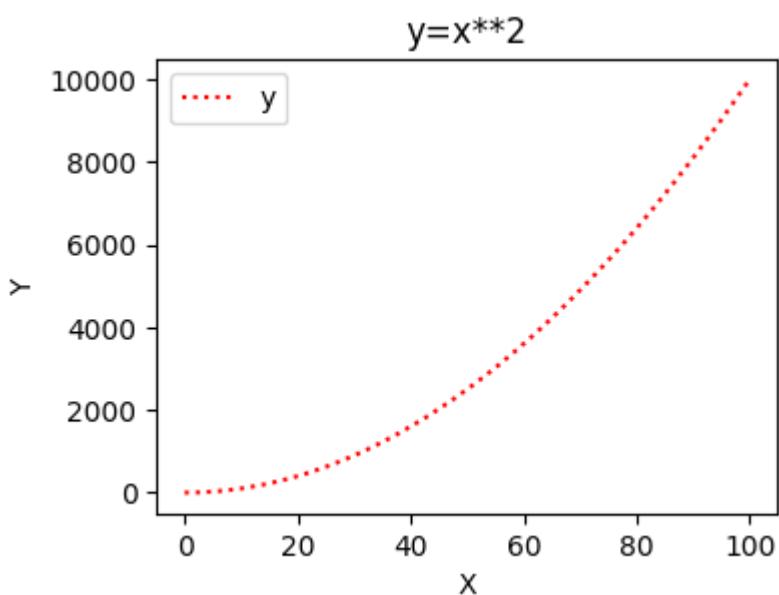
#在axe上绘制一条抛物线，红色 点
axe.plot(x,y,"r:")

axe.set_xlabel("X")
axe.set_ylabel("Y")
axe.legend(['y'])
```

```
axe.set_title("y=x**2")
plt.show()

创建两个子图并共用一个y轴
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)
plt.show()

创建四个子图并通过索引指定
fig, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
axes[0, 0].plot(x, y)
axes[1, 1].scatter(x, y)
plt.show()
```



## Images

You can use the `imshow` function to show images. Here is an example:

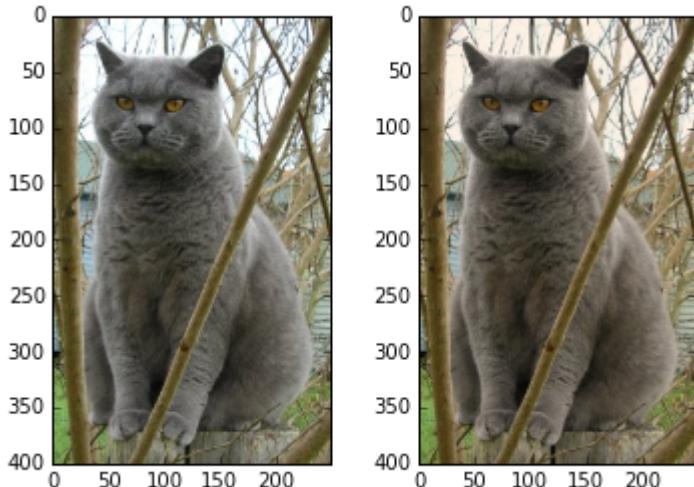
```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

Show the tinted image
plt.subplot(1, 2, 2)

A slight gotcha with imshow is that it might give strange results
if presented with data that is not uint8. To work around this, we
explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```



The following code will load an image from a file `image.png` and will display it as grayscale.

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

fname = 'image.png'
image = Image.open(fname).convert("L")
arr = np.asarray(image)
plt.imshow(arr, cmap='gray', vmin=0, vmax=255)
plt.show()
```

If you want to display the inverse grayscale, switch the `cmap` to `cmap='gray_r'`.

```

1、显示图片
import matplotlib.pyplot as plt #plt 用于显示图片
import matplotlib.image as mpimg #mpimg 用于读取图片
import numpy as np
lena = mpimg.imread('lena.png') #读取和代码处于同一目录下的lena.png
此时 lena 就已经是一个 np.array 了，可以对它进行任意处理
lena.shape #(512, 512, 3)
plt.imshow(lena) # 显示图片
plt.axis('off') # 不显示坐标轴
plt.show()

2、显示图片的第一个通道
lena_1 = lena[:, :, 0]
plt.imshow('lena_1')
plt.show()

此时会发现显示的是热量图，不是我们预想的灰度图，可以添加 cmap 参数，有如下几种添加方法：
#方法一
plt.imshow('lena_1', cmap='Greys_r')
plt.show()

#方法二
img = plt.imshow('lena_1')
img.set_cmap('gray') # 'hot' 是热量图
plt.show()

#3、将 RGB 转为灰度图
def rgb2gray(rgb):
 return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

gray = rgb2gray(lena)
也可以用 plt.imshow(gray, cmap = plt.get_cmap('gray'))
plt.imshow(gray, cmap='Greys_r')
plt.axis('off')
plt.show()

#4、对图像进行放缩
from scipy import misc
lena_new_sz = misc.imresize(lena, 0.5) # 第二个参数如果是整数，则为百分比，如果是tuple，则为输出图像的尺寸
plt.imshow(lena_new_sz)
plt.axis('off')
plt.show()

附上imresize的用法
功能：改变图像的大小。
用法：
B = imresize(A,m)
B = imresize(A,m,method)
B = imresize(A,[mrows ncols],method)
B = imresize(...,method,n)
B = imresize(...,method,h)

```

`imresize`函数使用由参数`method`指定的插值运算来改变图像的大小。  
`method`的几种可选值：  
'nearest' (默认值) 最近邻插值  
'bilinear' 双线性插值  
'bicubic' 双三次插值  
`B = imresize(A,m)`表示把图像A放大m倍  
`B = imresize(...,method,h)`中的h可以是任意一个FIR滤波器 (h通常由函数`ftrans2`、`fwind1`、`fwind2`、或`fsamp2`等生成的二维FIR滤波器)。

```
#5、保存 matplotlib 画出的图像
plt.savefig('lena_new_sz.png')
```

```
#5、将 array 保存为图像
from scipy import misc
misc.imsave('lena_new_sz.png', lena_new_sz)
```

```
#5、直接保存 array
#读取之后还是可以按照前面显示数组的方法对图像进行显示，这种方法完全不会对图像质量造成损失
np.save('lena_new_sz', lena_new_sz) # 会在保存的名字后面自动加上.npy
img = np.load('lena_new_sz.npy') # 读取前面保存的数组
```

更多图像显示参考：<https://www.cnblogs.com/denny402/p/5122594.html>

## PIL

```
#1、显示图片
from PIL import Image
im = Image.open('lena.png')
im.show()
```

```
#2、将 PIL Image 图片转换为 numpy 数组
im_array = np.array(im)
也可以用 np.asarray(im) 区别是 np.array() 是深拷贝，np.asarray() 是浅拷贝
```

```
#3、保存 PIL 图片
#直接调用 Image 类的 save 方法
```

```
from PIL import Image
I = Image.open('lena.png')
I.save('new_lena.png')
```

```
#4、将 numpy 数组转换为 PIL 图片
#这里采用 matplotlib.image 读入图片数组，注意这里读入的数组是 float32 型的，范围是 0-1，而 PIL.Image 数
据是 uint8 型的，范围是0-255，所以要进行转换：
import matplotlib.image as mpimg
from PIL import Image
```

```
lena = mpimg.imread('lena.png') # 这里读入的数据是 float32 型的，范围是0-1
im = Image.fromarray(np.uint8(lena*255))
im.show()
```

```
#5、RGB 转换为灰度图、二值化图
from PIL import Image
I = Image.open('lena.png')
I.show()
L = I.convert('L') #转化为灰度图
L = I.convert('1') #转化为二值化图
L.show()
```

附:PIL可以对图像的颜色进行转换，并支持诸如24位彩色、8位灰度图和二值图等模式，简单的转换可以通过Image.convert(mode)函数完成，其中mode表示输出的颜色模式，例如'‘L’’表示灰度，‘‘1’’表示二值图模式等。但是利用convert函数将灰度图转换为二值图时，是采用固定的阈值127来实现的，即灰度高于127的像素值为1，而灰度低于127的像素值为0。

<http://codingpy.com/article/an-introduction-to-numpy/>

<http://codingpy.com/article/a-quick-intro-to-matplotlib/>

<http://codingpy.com/article/a-quick-intro-to-pandas/>

## Pandas

Pandas 是我最喜爱的库之一。通过带有标签的列和索引，Pandas 使我们可以以一种所有人都能理解的方式来处理数据。它可以帮助我们毫不费力地从诸如 csv 类型的文件中导入数据。我们可以用它快速地对数据进行复杂的转换和过滤等操作。Pandas 真是超级棒。

我觉得它和 Numpy、Matplotlib 一起构成了一个 Python 数据探索和分析的强大基础。Scipy（将会在下一篇推文里介绍）当然也是一大主力并且是一个绝对赞的库，但是我觉得前三者才是 Python 科学计算真正的顶梁柱。

### 安装pandas:

```
pip install pandas
```

### 导入 Pandas:

第一件事当然是请出我们的明星 —— Pandas。

```
import pandas as pd # This is the standard
```

这是导入 pandas 的标准方法。我们不想一直写 pandas 的全名，但是保证代码的简洁和避免命名冲突都很重要，所以折中使用 pd。如果你去看别人使用 pandas 的代码，就会看到这种导入方式。

### Pandas 中的数据类型:

Pandas 基于两种数据类型，series 和 dataframe。

series 是一种一维的数据类型，其中的每个元素都有各自的标签。如果你之前看过这个系列关于 [Numpy](#) 的推文，你可以把它当作一个由带标签的元素组成的 `numpy` 数组。标签可以是数字或者字符。

dataframe 是一个二维的、表格型的数据结构。Pandas 的 dataframe 可以储存许多不同类型的数据，并且每个轴都有标签。你可以把它当作一个 series 的字典。

## 将数据导入 Pandas:

在对数据进行修改、探索和分析之前，我们得先导入数据。多亏了 Pandas，这比在 `Numpy` 中还要容易。

这里我鼓励你去找到自己感兴趣的数据并用来练习。你的（或者别的）国家的网站就是不错的数据源。如果要举例的话，首推[英国政府数据](#)和[美国政府数据](#)。[Kaggle](#)也是个很好的数据源。

我将使用英国降雨数据，这个数据集可以很容易地从英国政府网站上下载到。此外，我还下载了一些日本降雨量的数据。

英国降雨数据：[下载地址](#) 日本的数据实在是没找到，抱歉。

```
Reading a csv into Pandas.
df = pd.read_csv('uk_rain_2014.csv', header=0)
df.shape # 返回数据的维度
```

译者注：如果你的数据集中有中文的话，最好在里面加上 `encoding = 'gbk'`，以避免乱码问题。后面的导出数据的时候也一样。

这里我们从 `csv` 文件里导入了数据，并储存在 `dataframe` 中。这一步非常简单，你只需要调用 `read_csv` 然后将文件的路径传进去就行了。`header` 关键字告诉 Pandas 哪些是数据的列名。如果没有列名的话就将它设定为 `None`。Pandas 非常聪明，所以这个经常可以省略。

## 探索和分析的数据:

现在数据已经导入到 Pandas 了，我们也许想看一眼数据来得到一些基本信息，以便在真正开始探索之前找到一些方向。

查看前 x 行的数据：

```
Getting first x rows.
df.head(5) # 默认是前5行
```

我们只需要调用 `head()` 函数并且将想要查看的行数传入。

得到的结果如下：

|   | Water Year | Rain (mm) Oct-Sep | Outflow (m3/s) Oct-Sep | Rain (mm) Dec-Feb | Outflow (m3/s) Dec-Feb | Rain (mm) Jun-Aug | Outflow (m3/s) Jun-Aug |
|---|------------|-------------------|------------------------|-------------------|------------------------|-------------------|------------------------|
| 0 | 1980/81    | 1182              | 5408                   | 292               | 7248                   | 174               | 2212                   |
| 1 | 1981/82    | 1098              | 5112                   | 257               | 7316                   | 242               | 1936                   |
| 2 | 1982/83    | 1156              | 5701                   | 330               | 8567                   | 124               | 1802                   |
| 3 | 1983/84    | 993               | 4265                   | 391               | 8905                   | 141               | 1078                   |
| 4 | 1984/85    | 1182              | 5364                   | 217               | 5813                   | 343               | 4313                   |

你可能还想看看最后几行：

```
Getting last x rows.
df.tail(5)
```

跟 `head` 一样，我们只需要调用 `tail` 并且传入想要查看的行数即可。注意，它并不是从最后一行倒着显示的，而是按照数据原来的顺序显示。

得到的结果如下：

|    | Water Year | Rain (mm) Oct-Sep | Outflow (m3/s) Oct-Sep | Rain (mm) Dec-Feb | Outflow (m3/s) Dec-Feb | Rain (mm) Jun-Aug | Outflow (m3/s) Jun-Aug |
|----|------------|-------------------|------------------------|-------------------|------------------------|-------------------|------------------------|
| 28 | 2008/09    | 1139              | 4941                   | 268               | 6690                   | 323               | 3189                   |
| 29 | 2009/10    | 1103              | 4738                   | 255               | 6435                   | 244               | 1958                   |
| 30 | 2010/11    | 1053              | 4521                   | 265               | 6593                   | 267               | 2885                   |
| 31 | 2011/12    | 1285              | 5500                   | 339               | 7630                   | 379               | 5261                   |
| 32 | 2012/13    | 1090              | 5329                   | 350               | 9615                   | 187               | 1797                   |

你通常使用列的名字来在 Pandas 中查找列。这一点很好而且易于使用，但是有时列名太长，比如调查问卷的一整个问题。不过你把列名缩短之后一切就好说了。

```
Changing column labels.
df.columns = ['water_year', 'rain_octsep', 'outflow_octsep',
 'rain_decfeb', 'outflow_decfeb', 'rain_junaug', 'outflow_junaug']

df.head(5)
```

需要注意的一点是，我故意没有在每列的标签中使用空格和破折号。之后你会看到这样为变量命名可以使我们少打一些字符。

你得到的数据与之前的一样，只是换了列的名字：

|   | water_year | rain_octsep | outflow_octsep | rain_decfeb | outflow_decfeb | rain_junaug | outflow_junaug |
|---|------------|-------------|----------------|-------------|----------------|-------------|----------------|
| 0 | 1980/81    | 1182        | 5408           | 292         | 7248           | 174         | 2212           |
| 1 | 1981/82    | 1098        | 5112           | 257         | 7316           | 242         | 1936           |
| 2 | 1982/83    | 1156        | 5701           | 330         | 8567           | 124         | 1802           |
| 3 | 1983/84    | 993         | 4265           | 391         | 8905           | 141         | 1078           |
| 4 | 1984/85    | 1182        | 5364           | 217         | 5813           | 343         | 4313           |

你通常会想知道数据的另一个特征——它有多少条记录。在 Pandas 中，一条记录对应着一行，所以我们可以对数据集调用 `len` 方法，它将返回数据集的总行数：

```
Finding out how many rows dataset has.
len(df)
```

上面的代码返回一个表示数据行数的整数，在我的数据集中，这个值是 33。

你可能还想知道数据集的一些基本的统计数据，在 Pandas 中，这个操作简单到哭：

```
Finding out basic statistical information on your dataset.
pd.options.display.float_format = '{:.3f}'.format # Limit output to 3 decimal places.
df.describe()
```

这将返回一张表，其中有诸如总数、均值、标准差之类的统计数据：

|       | rain_octsep | outflow_octsep | rain_decfeb | outflow_decfeb | rain_junaug | outflow_junaug |
|-------|-------------|----------------|-------------|----------------|-------------|----------------|
| count | 33.000      | 33.000         | 33.000      | 33.000         | 33.000      | 33.000         |
| mean  | 1,129.000   | 5,019.182      | 325.364     | 7,926.545      | 237.485     | 2,439.758      |
| std   | 101.900     | 658.588        | 69.995      | 1,692.800      | 66.168      | 1,025.914      |
| min   | 856.000     | 3,479.000      | 206.000     | 4,578.000      | 103.000     | 1,078.000      |
| 25%   | 1,053.000   | 4,506.000      | 268.000     | 6,690.000      | 193.000     | 1,797.000      |
| 50%   | 1,139.000   | 5,112.000      | 309.000     | 7,630.000      | 229.000     | 2,142.000      |
| 75%   | 1,182.000   | 5,497.000      | 360.000     | 8,905.000      | 280.000     | 2,959.000      |
| max   | 1,387.000   | 6,391.000      | 484.000     | 11,486.000     | 379.000     | 5,261.000      |

## 过滤:

在探索数据的时候，你可能经常想要抽取数据中特定的样本，比如你有一个关于工作满意度的调查表，你可能就想要提取特定行业或者年龄的人的数据。

在 Pandas 中有多种方法可以实现提取我们想要的信息：

有时你想提取一整列，使用列的标签可以非常简单地做到：

```
Getting a column by label
df['rain_octsep']
```

注意，当我们提取列的时候，会得到一个 series，而不是 dataframe。记得我们前面提到过，你可以把 dataframe 看作是一个 series 的字典(key是列名)，所以在抽取列的时候，我们就会得到一个 series。

还记得我在命名列标签的时候特意指出的吗？不用空格、破折号之类的符号，这样我们就可以像访问对象属性一样访问数据集的列——只用一个点号。

```
Getting a column by label using .
df.rain_octsep
```

这句代码返回的结果与前一个例子完全一样——是我们选择的那列数据。

如果你读过这个系列关于 Numpy 的推文，你可能还记得一个叫做 **布尔过滤 (boolean masking)** 的技术，通过在一个数组上运行条件来得到一个布尔数组。在 Pandas 里也可以做到。

```
Creating a series of booleans based on a conditional
df.rain_octsep < 1000 # Or df['rain_octsep'] < 1000
```

上面的代码将会返回一个由布尔值构成的 dataframe。`True` 表示在十月-九月降雨量小于 1000 mm，`False` 表示大于等于 1000 mm。

我们可以用这些条件表达式来过滤现有的 dataframe。

```
Using a series of booleans to filter
df[df.rain_octsep < 1000]
```

这条代码只返回十月-九月降雨量小于 1000 mm 的记录：

|    | water_year | rain_octsep | outflow_octsep | rain_decfeb | outflow_decfeb | rain_junaug | outflow_junaug |
|----|------------|-------------|----------------|-------------|----------------|-------------|----------------|
| 3  | 1983/84    | 993         | 4265           | 391         | 8905           | 141         | 1078           |
| 8  | 1988/89    | 976         | 4330           | 309         | 6465           | 200         | 1440           |
| 15 | 1995/96    | 856         | 3479           | 245         | 5515           | 172         | 1439           |

也可以通过复合条件表达式来进行过滤：

```
Filtering by multiple conditionals
df[(df.rain_octsep < 1000) & (df.outflow_octsep < 4000)] # Can't use the keyword 'and'
```

这条代码只会返回 `rain_octsep` 小于 1000 的和 `outflow_octsep` 小于 4000 的记录：

注意重要的一点：这里不能用 `and` 关键字，因为会引发操作顺序的问题。必须用 `&` 和圆括号。

|    | water_year | rain_octsep | outflow_octsep | rain_decfeb | outflow_decfeb | rain_junaug | outflow_junaug |
|----|------------|-------------|----------------|-------------|----------------|-------------|----------------|
| 15 | 1995/96    | 856         | 3479           | 245         | 5515           | 172         | 1439           |

如果你的数据中字符串，好消息，你也可以使用字符串方法来进行过滤：

```
Filtering by string methods
df[df.water_year.str.startswith('199')]
```

注意，你必须用 `.str.[string method]`，而不能直接在字符串上调用字符方法。上面的代码返回所有 90 年代的记录：

|    | water_year | rain_octsep | outflow_octsep | rain_decfeb | outflow_decfeb | rain_junaug | outflow_junaug |
|----|------------|-------------|----------------|-------------|----------------|-------------|----------------|
| 10 | 1990/91    | 1022        | 4418           | 305         | 7120           | 216         | 1923           |
| 11 | 1991/92    | 1151        | 4506           | 246         | 5493           | 280         | 2118           |
| 12 | 1992/93    | 1130        | 5246           | 308         | 8751           | 219         | 2551           |
| 13 | 1993/94    | 1162        | 5583           | 422         | 10109          | 193         | 1638           |
| 14 | 1994/95    | 1110        | 5370           | 484         | 11486          | 103         | 1231           |
| 15 | 1995/96    | 856         | 3479           | 245         | 5515           | 172         | 1439           |
| 16 | 1996/97    | 1047        | 4019           | 258         | 5770           | 256         | 2102           |
| 17 | 1997/98    | 1169        | 4953           | 341         | 7747           | 285         | 3206           |
| 18 | 1998/99    | 1268        | 5824           | 360         | 8771           | 225         | 2240           |
| 19 | 1999/00    | 1204        | 5665           | 417         | 10021          | 197         | 2166           |

索引：

之前的部分展示了如何通过列操作来得到数据，但是 Pandas 的行也有标签。行标签可以是基于数字的或者是标签，而且获取行数据的方法也根据标签的类型各有不同。

如果你的行标签是数字型的，你可以通过 `iloc` 来引用：

```
Getting a row via a numerical index
df.iloc[30] # 这是df中的索引，从0开始，对应csv文件中的第32行
```

`iloc` 只对数字型的标签有用。它会返回给定行的 series，行中的每一列都是返回 series 的一个元素。

也许你的数据集中有年份或者年龄的列，你可能想通过这些年份或者年龄来引用行，这个时候我们就可以设置一个（或者多个）新的索引：

```
Setting a new index from an existing column
df = df.set_index(['water_year'])
df.head(5)
```

上面的代码将 `water_year` 列设置为索引。注意，列的名字实际上是一个列表，虽然上面的例子中只有一个元素。如果你想设置多个索引，只需要在列表中加入列的名字即可。

|            | rain_octsep | outflow_octsep | rain_decfeb | outflow_decfeb | rain_junaug | outflow_junaug |
|------------|-------------|----------------|-------------|----------------|-------------|----------------|
| water_year |             |                |             |                |             |                |
| 1980/81    | 1182        | 5408           | 292         | 7248           | 174         | 2212           |
| 1981/82    | 1098        | 5112           | 257         | 7316           | 242         | 1936           |
| 1982/83    | 1156        | 5701           | 330         | 8567           | 124         | 1802           |
| 1983/84    | 993         | 4265           | 391         | 8905           | 141         | 1078           |
| 1984/85    | 1182        | 5364           | 217         | 5813           | 343         | 4313           |

上例中我们设置的索引列中都是字符型数据，这意味着我们不能继续使用 `iloc` 来引用，那我们用什么呢？用 `loc`。

```
Getting a row via a label-based index
df.loc['2000/01']
```

和 `iloc` 一样，`loc` 会返回你引用的列，唯一一点不同就是此时你使用的是基于字符串的引用，而不是基于数字的。

还有一个引用列的常用常用方法——`ix`。如果 `loc` 是基于标签的，而 `iloc` 是基于数字的，那 `ix` 是基于什么的？事实上，`ix` 是基于标签的查询方法，但它同时也支持数字型索引作为备选。

```
Getting a row via a label-based or numerical index
df.ix['1999/00'] # Label based with numerical index fallback *Not recommended
```

与 `iloc`、`loc` 一样，它也会返回你查询的行。

如果 `ix` 可以同时起到 `loc` 和 `iloc` 的作用，那为什么还要用后两个？一大原因就是 `ix` 具有轻微的不可预测性。还记得我说过它所支持的数字型索引只是备选吗？这一特性可能会导致 `ix` 产生一些奇怪的结果，比如讲一个数字解释为一个位置。而使用 `iloc` 和 `loc` 会很安全、可预测并且让人放心。但是我要指出的是，`ix` 比 `iloc` 和 `loc` 要快一些。

将索引排序通常会很有用，在 Pandas 中，我们可以对 dataframe 调用 `sort_index` 方法进行排序。

```
df.sort_index(ascending=False).head(5) #inplace=True to apply the sorting in place
```

我的索引本来就是有序的，为了演示，我将参数 `ascending` 设置为 `false`，这样我的数据就会呈降序排列。

|                         | <code>rain_octsep</code> | <code>outflow_octsep</code> | <code>rain_decfeb</code> | <code>outflow_decfeb</code> | <code>rain_junaug</code> | <code>outflow_junaug</code> |
|-------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|
| <code>water_year</code> |                          |                             |                          |                             |                          |                             |
| <code>2012/13</code>    | 1090                     | 5329                        | 350                      | 9615                        | 187                      | 1797                        |
| <code>2011/12</code>    | 1285                     | 5500                        | 339                      | 7630                        | 379                      | 5261                        |
| <code>2010/11</code>    | 1053                     | 4521                        | 265                      | 6593                        | 267                      | 2885                        |
| <code>2009/10</code>    | 1103                     | 4738                        | 255                      | 6435                        | 244                      | 1958                        |
| <code>2008/09</code>    | 1139                     | 4941                        | 268                      | 6690                        | 323                      | 3189                        |

当你将一列设置为索引的时候，它就不再是数据的一部分了。如果你想将索引恢复为数据，调用 `set_index` 相反的方法 `reset_index` 即可：

```
Returning an index to data
df = df.reset_index('water_year')
df.head(5)
```

这一语句会将索引恢复成数据形式：

|   | <code>water_year</code> | <code>rain_octsep</code> | <code>outflow_octsep</code> | <code>rain_decfeb</code> | <code>outflow_decfeb</code> | <code>rain_junaug</code> | <code>outflow_junaug</code> |
|---|-------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|
| 0 | 1980/81                 | 1182                     | 5408                        | 292                      | 7248                        | 174                      | 2212                        |
| 1 | 1981/82                 | 1098                     | 5112                        | 257                      | 7316                        | 242                      | 1936                        |
| 2 | 1982/83                 | 1156                     | 5701                        | 330                      | 8567                        | 124                      | 1802                        |
| 3 | 1983/84                 | 993                      | 4265                        | 391                      | 8905                        | 141                      | 1078                        |
| 4 | 1984/85                 | 1182                     | 5364                        | 217                      | 5813                        | 343                      | 4313                        |

对数据集应用函数：

有时你想对数据集中的数据进行改变或者某种操作。比方说，你有一列年份的数据，你需要新的一列来表示这些年份对应的年代。Pandas 中有两个非常有用的函数，`apply` 和 `applymap`。

```

Applying a function to a column
def base_year(year):
 base_year = year[:4]
 base_year= pd.to_datetime(base_year).year
 return base_year

df['year'] = df.water_year.apply(base_year)
df.head(5)

```

上面的代码创建了一个叫做 `year` 的列，它只将 `water_year` 列中的年提取了出来。这就是 `apply` 的用法，即对一列数据应用函数。如果你想对整个数据集应用函数，就要使用 `applymap`。

|          | <code>water_year</code> | <code>rain_octsep</code> | <code>outflow_octsep</code> | <code>rain_decfeb</code> | <code>outflow_decfeb</code> | <code>rain_junaug</code> | <code>outflow_junaug</code> | <code>year</code> |
|----------|-------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|-------------------|
| <b>0</b> | 1980/81                 | 1182                     | 5408                        | 292                      | 7248                        | 174                      | 2212                        | 1980              |
| <b>1</b> | 1981/82                 | 1098                     | 5112                        | 257                      | 7316                        | 242                      | 1936                        | 1981              |
| <b>2</b> | 1982/83                 | 1156                     | 5701                        | 330                      | 8567                        | 124                      | 1802                        | 1982              |
| <b>3</b> | 1983/84                 | 993                      | 4265                        | 391                      | 8905                        | 141                      | 1078                        | 1983              |
| <b>4</b> | 1984/85                 | 1182                     | 5364                        | 217                      | 5813                        | 343                      | 4313                        | 1984              |

### 操作数据集的结构：

另一常见的做法是重新建立数据结构，使得数据集呈现出一种更方便并且（或者）有用的形式。

掌握这些转换最简单的方法就是观察转换的过程。比起这篇文章的其他部分，接下来的操作需要你跟着练习以便能掌握它们。

首先，是 `groupby`：

```

#Manipulating structure (groupby, unstack, pivot)
Groupby
df.groupby(df.year // 10 *10).max()

```

`groupby` 会按照你选择的列对数据集进行分组。上例是按照年代分组。不过仅仅这样做并没有什么用，我们必须对其调用函数，比如 `max`、`min`、`mean` 等等。例中，我们可以得到 90 年代的均值。

|             | <code>water_year</code> | <code>rain_octsep</code> | <code>outflow_octsep</code> | <code>rain_decfeb</code> | <code>outflow_decfeb</code> | <code>rain_junaug</code> | <code>outflow_junaug</code> | <code>year</code> |
|-------------|-------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|--------------------------|-----------------------------|-------------------|
| <b>year</b> |                         |                          |                             |                          |                             |                          |                             |                   |
| <b>1980</b> | 1989/90                 | 1210                     | 5701                        | 470                      | 10520                       | 343                      | 4313                        | 1989              |
| <b>1990</b> | 1999/00                 | 1268                     | 5824                        | 484                      | 11486                       | 285                      | 3206                        | 1999              |
| <b>2000</b> | 2009/10                 | 1387                     | 6391                        | 437                      | 10926                       | 357                      | 5168                        | 2009              |
| <b>2010</b> | 2012/13                 | 1285                     | 5500                        | 350                      | 9615                        | 379                      | 5261                        | 2012              |

你也可以按照多列进行分组：

```
Grouping by multiple columns
decade_rain = df.groupby([df.year // 10 * 10, df.rain_octsep // 1000 * 1000])
[['outflow_octsep',
 'outflow_decfeb', 'outflow_junaug']].mean()
decade_rain
```

|      |             | outflow_octsep | outflow_decfeb | outflow_junaug |
|------|-------------|----------------|----------------|----------------|
| year | rain_octsep |                |                |                |
| 1980 | 0           | 4297.500000    | 7685.000000    | 1259.000000    |
|      | 1000        | 5289.625000    | 7933.000000    | 2572.250000    |
| 1990 | 0           | 3479.000000    | 5515.000000    | 1439.000000    |
|      | 1000        | 5064.888889    | 8363.111111    | 2130.555556    |
| 2000 | 1000        | 5030.800000    | 7812.100000    | 2685.900000    |
| 2010 | 1000        | 5116.666667    | 7946.000000    | 3314.333333    |

接下来是 `unstack`，最开始可能有一些困惑，它可以将一列数据设置为列标签。最好还是看看实际的操作：

```
Unstacking
decade_rain.unstack(0)
```

这条语句将上例中的 dataframe 转换为下面的形式。它将第 0 列，也就是 `year` 列设置为列的标签。

| year        | outflow_octsep |             |        |             | outflow_decfeb |             |        |        | outflow_junaug |             |        |             |
|-------------|----------------|-------------|--------|-------------|----------------|-------------|--------|--------|----------------|-------------|--------|-------------|
|             | 1980           | 1990        | 2000   | 2010        | 1980           | 1990        | 2000   | 2010   | 1980           | 1990        | 2000   | 2010        |
| rain_octsep |                |             |        |             |                |             |        |        |                |             |        |             |
| 0           | 4297.500       | 3479.000000 | NaN    | NaN         | 7685.0         | 5515.000000 | NaN    | NaN    | 1259.00        | 1439.000000 | NaN    | NaN         |
| 1000        | 5289.625       | 5064.888889 | 5030.8 | 5116.666667 | 7933.0         | 8363.111111 | 7812.1 | 7946.0 | 2572.25        | 2130.555556 | 2685.9 | 3314.333333 |

让我们再操作一次。这次使用第 1 列，也就是 `rain_octsep` 列：

```
More unstacking
decade_rain.unstack(1)
```

|             | outflow_octsep |             |        |             | outflow_decfeb |             |        |        | outflow_junaug |             |        |             |
|-------------|----------------|-------------|--------|-------------|----------------|-------------|--------|--------|----------------|-------------|--------|-------------|
| year        | 1980           | 1990        | 2000   | 2010        | 1980           | 1990        | 2000   | 2010   | 1980           | 1990        | 2000   | 2010        |
| rain_octsep |                |             |        |             |                |             |        |        |                |             |        |             |
| 0           | 4297.500       | 3479.000000 | NaN    | NaN         | 7685.0         | 5515.000000 | NaN    | NaN    | 1259.00        | 1439.000000 | NaN    | NaN         |
| 1000        | 5289.625       | 5064.888889 | 5030.8 | 5116.666667 | 7933.0         | 8363.111111 | 7812.1 | 7946.0 | 2572.25        | 2130.555556 | 2685.9 | 3314.333333 |

在进行下次操作之前，我们先创建一个用于演示的 dataframe：

```
Create a new dataframe containing entries which
has rain_octsep values of greater than 1250
high_rain = df[df.rain_octsep > 1250]
high_rain
```

上面的代码将会产生如下的 dataframe，我们将会在上面演示轴向旋转（pivoting）。

|           | <b>water_year</b> | <b>rain_octsep</b> | <b>outflow_octsep</b> | <b>rain_decfeb</b> | <b>outflow_decfeb</b> | <b>rain_junaug</b> | <b>outflow_junaug</b> | <b>year</b> |
|-----------|-------------------|--------------------|-----------------------|--------------------|-----------------------|--------------------|-----------------------|-------------|
| <b>18</b> | 1998/99           | 1268               | 5824                  | 360                | 8771                  | 225                | 2240                  | 1998        |
| <b>26</b> | 2006/07           | 1387               | 6391                  | 437                | 10926                 | 357                | 5168                  | 2006        |
| <b>31</b> | 2011/12           | 1285               | 5500                  | 339                | 7630                  | 379                | 5261                  | 2011        |

轴旋转其实就是我们之前已经看到的那些操作的一个集合。首先，它会设置一个新的索引（`set_index()`），然后对索引排序（`sort_index()`），最后调用 `unstack`。以上的步骤合在一起就是 `pivot`。接下来看看你能不能搞清楚下面的代码在干什么：

```
#Pivoting
#does set_index, sort_index and unstack in a row
high_rain.pivot('year', 'rain_octsep')[['outflow_octsep', 'outflow_decfeb',
'outflow_junaug']].fillna('')
```

注意，最后一个 `.fillna('')`。`pivot` 产生了很多空的记录，也就是值为 `NaN` 的记录。我个人觉得数据集里面有很多 `NaN` 会很烦，所以使用了 `fillna('')`。你也可以用别的别的东西，比方说 0。我们也可以使用 `dropna(how = 'any')` 来删除有 `NaN` 的行，不过这样就把所有的数据都删掉了，所以不这样做。

|                    | <b>outflow_octsep</b> |             |             | <b>outflow_decfeb</b> |             |             | <b>outflow_junaug</b> |             |             |
|--------------------|-----------------------|-------------|-------------|-----------------------|-------------|-------------|-----------------------|-------------|-------------|
| <b>rain_octsep</b> | <b>1268</b>           | <b>1285</b> | <b>1387</b> | <b>1268</b>           | <b>1285</b> | <b>1387</b> | <b>1268</b>           | <b>1285</b> | <b>1387</b> |
| <b>year</b>        |                       |             |             |                       |             |             |                       |             |             |
| <b>1998</b>        | 5824                  |             |             | 8771                  |             |             | 2240                  |             |             |
| <b>2006</b>        |                       |             | 6391        |                       |             | 10926       |                       |             | 5168        |
| <b>2011</b>        |                       | 5500        |             |                       | 7630        |             |                       | 5261        |             |

上面的 dataframe 展示了所有降雨超过 1250 的 `outflow`。诚然，这并不是讲解 `pivot` 实际应用最好的例子，但希望你能明白它的意思。看看你能在你的数据集上得到什么结果。

## 合并数据集：

有时你有两个相关联的数据集，你想将它们放在一起比较或者合并它们。好的，没问题，在 Pandas 里很简单：

```
Merging two datasets together
rain_jpn = pd.read_csv('jpn_rain.csv')
rain_jpn.columns = ['year', 'jpn_rainfall']

uk_jpn_rain = df.merge(rain_jpn, on='year')
uk_jpn_rain.head(5)
```

首先你需要通过 `on` 关键字来指定需要合并的列。通常你可以省略这个参数，Pandas 将会自动选择要合并的列。

如下图所示，两个数据集在年份这一类上合并了。`jpn_rain` 数据集只有年份和降雨量两列，通过年份列合并之后，`jpn_rain` 中只有降雨量那一列合并到了 `UK_rain` 数据集中。

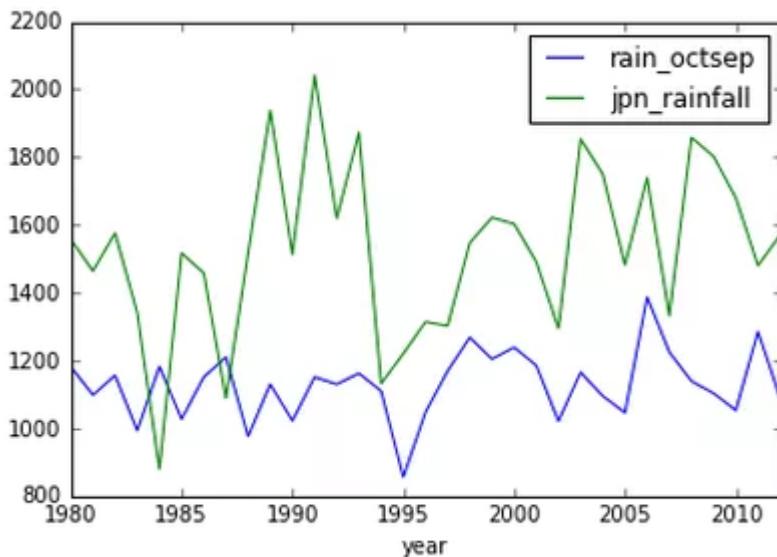
|   | water_year | rain_octsep | outflow_octsep | rain_decfab | outflow_decfab | rain_junaug | outflow_junaug | year | jpn_rainfall |
|---|------------|-------------|----------------|-------------|----------------|-------------|----------------|------|--------------|
| 0 | 1980/81    | 1182        | 5408           | 292         | 7248           | 174         | 2212           | 1980 | 1558         |
| 1 | 1981/82    | 1098        | 5112           | 257         | 7316           | 242         | 1936           | 1981 | 1464         |
| 2 | 1982/83    | 1156        | 5701           | 330         | 8567           | 124         | 1802           | 1982 | 1576         |
| 3 | 1983/84    | 993         | 4265           | 391         | 8905           | 141         | 1078           | 1983 | 1341         |
| 4 | 1984/85    | 1182        | 5364           | 217         | 5813           | 343         | 4313           | 1984 | 880          |

## 使用 Pandas 快速作图:

Matplotlib 很棒，但是想要绘制出还算不错的图表却要写不少代码，而有时你只是想粗略的做个图来探索下数据，搞清楚数据的含义。Pandas 通过 `plot` 来解决这个问题：

```
Using pandas to quickly plot graphs
uk_jpn_rain.plot(x='year', y=['rain_octsep', 'jpn_rainfall'])
```

这会调用 Matplotlib 快速轻松地绘出了你的数据图。通过这个图你就可以在视觉上分析数据，而且它能在探索数据的时候给你一些方向。比如，看到我的数据图，你会发现在 1995 年的英国好像有一场干旱。



你会发现英国的降雨明显少于日本，但人们却说英国总是下雨。

## 保存你的数据集:

在清洗、重塑、探索完数据之后，你最后的数据集可能会发生很大改变，并且比最开始的时候更有用。你应该保存原始的数据集，但是你同样应该保存处理之后的数据。

```
Saving your data to a csv
df.to_csv('uk_rain.csv')
```

上面的代码将会保存你的数据到 `csv` 文件以便下次使用。

我们对 Pandas 的介绍就到此为止了。就像我之前所说的，Pandas 非常强大，我们只是领略到了一点皮毛而已，不过你现在知道的应该足够你开始清洗和探索数据了。

像以前一样，我建议你用自己感兴趣的数据集做一下练习，坐下来，一杯啤酒配数据。这确实是你唯一熟悉 Pandas 以及这个系列其他库的方式。而且你也许会发现一些有趣的东西。

## 3. Advanced Tutorial

### 类

self

<https://www.jianshu.com/p/bdbd577314f9>

每次调用内部的方法时，方法前面加 self.

举例：

```
class MyClass:
 def __init__(self):
 pass
 def func1(self):
 # do something
 print('a') #for example
 self.common_func()
 def func2(self):
 # do something
 self.common_func()

 def common_func(self):
 pass
```

- 多线程

问题：

- 1、Python 多线程为什么耗时更长？
- 2、为什么在 Python 里面推荐使用多进程而不是多线程？

## 1 基础知识

现在的 PC 都是多核的，使用多线程能充分利用 CPU 来提供程序的执行效率。

### 1.1 线程

线程是一个基本的 CPU 执行单元。它必须依托于进程存活。一个线程是一个**execution context**（执行上下文），即一个 CPU 执行时所需要的一串指令。

## 1.2 进程

进程是指一个程序在给定数据集合上的一次执行过程，是系统进行资源分配和运行调用的独立单位。可以简单地理解为操作系统中正在执行的程序。也就是说，每个应用程序都有一个自己的进程。

每一个进程启动时都会最先产生一个线程，即主线程。然后主线程会再创建其他的子线程。

## 1.3 两者区别

- 线程必须在某个进程中执行。
- 一个进程可包含多个线程，其中有且只有一个主线程。
- 多线程共享同一个地址空间、打开的文件以及其他资源。
- 多进程共享物理内存、磁盘、打印机以及其他资源。

## 1.4 线程的类型

线程的作用可以划分为不同的类型。大致可分为：

- 主线程
- 子线程
- 守护线程（后台线程）
- 前台线程

# 2 Python 多线程

## 2.1 GIL

其他语言，CPU 是多核时是支持多个线程同时执行。但在 Python 中，无论是单核还是多核，同时只能由一个线程在执行。其根源是 GIL 的存在。

GIL 的全称是 Global Interpreter Lock(全局解释器锁)，来源是 Python 设计之初的考虑，为了数据安全所做的决定。某个线程想要执行，必须先拿到 GIL，我们可以把 GIL 看作是“通行证”，并且在一个 Python 进程中，GIL 只有一个。拿不到通行证的线程，就不允许进入 CPU 执行。

而目前 Python 的解释器有多种，例如：

- **CPython**: CPython 是用C语言实现的 Python 解释器。作为官方实现，它是最广泛使用的 Python 解释器。
- **PyPy**: PyPy 是用RPython实现的解释器。RPython 是 Python 的子集，具有静态类型。这个解释器的特点是即时编译，支持多重后端（C, CLI, JVM）。PyPy 旨在提高性能，同时保持最大兼容性（参考 CPython 的实现）。
- **Jython**: Jython 是一个将 Python 代码编译成 Java 字节码的实现，运行在JVM (Java Virtual Machine) 上。另外，它可以像使用 Python 模块一样，导入并使用任何Java类。
- **IronPython**: IronPython 是一个针对 .NET 框架的 Python 实现。它可以用 Python 和 .NET framework 的库，也能将 Python 代码暴露给 .NET 框架中的其他语言。

GIL 只在 CPython 中才有，而在 PyPy 和 Jython 中是没有 GIL 的。

每次释放 GIL 锁，线程进行锁竞争、切换线程，会消耗资源。这就导致打印线程执行时长，会发现耗时更长的原因。

并且由于 GIL 锁存在，Python 里一个进程永远只能同时执行一个线程（拿到 GIL 的线程才能执行），这就是为什么在多核CPU上，Python 的多线程效率并不高的根本原因。

## 2.2 创建多线程

Python提供两个模块进行多线程的操作，分别是 `thread` 和 `threading`，前者是比较低级的模块，用于更底层的操作，一般应用级别的开发不常用。

- 方法1：直接使用 `threading.Thread()`

```
import threading

这个函数名可随便定义
def run(n):
 print("current task: ", n)

if __name__ == "__main__":
 t1 = threading.Thread(target=run, args=("thread 1",))
 t2 = threading.Thread(target=run, args=("thread 2",))
 t1.start()
 t2.start()
```

- 方法2：继承 `threading.Thread` 来自定义线程类，重写 `run` 方法

```
import threading

class MyThread(threading.Thread):
 def __init__(self, n):
 super(MyThread, self).__init__() # 重构run函数必须要写
 self.n = n

 def run(self):
 print("current task: ", n)

if __name__ == "__main__":
 t1 = MyThread("thread 1")
 t2 = MyThread("thread 2")

 t1.start()
 t2.start()
```

## 2.3 线程合并

`Join` 函数执行顺序是逐个执行每个线程，执行完毕后继续往下执行。主线程结束后，子线程还在运行，`join` 函数使得主线程等到子线程结束时才退出。

```
import threading

def count(n):
 while n > 0:
 n -= 1

if __name__ == "__main__":
 t1 = threading.Thread(target=count, args=("100000",))
 t2 = threading.Thread(target=count, args=("100000",))
```

```
t1.start()
t2.start()
将 t1 和 t2 加入到主线程中
t1.join()
t2.join()
```

## 2.4 线程同步与互斥锁

线程之间数据共享的。当多个线程对某一个共享数据进行操作时，就需要考虑到线程安全问题。`threading` 模块中定义了`Lock` 类，提供了互斥锁的功能来保证多线程情况下数据的正确性。

用法的基本步骤：

```
#创建锁
mutex = threading.Lock()
#锁定
mutex.acquire([timeout])
#释放
mutex.release()
```

其中，锁定方法`acquire`可以有一个超时时间的可选参数`timeout`。如果设定了`timeout`，则在超时后通过返回值可以判断是否得到了锁，从而可以进行一些其他的处理。

`acquire` 锁定某个线程后，即使这个线程耗时非常久，后面的线程也必须等待其把这个线程执行完毕后才会继续执行（特别是对数据的操作完成后），这样就保证了对数据的顺序操作：

具体用法见示例代码：

```
#!/usr/bin/env python

import threading
import time

num = 0
delay = 5
mutex = threading.Lock()

class MyThread(threading.Thread):
 def run(self):
 global num
 global delay

 if mutex.acquire(1):
 num = num + 1
 print("now the num is:", num)
 time.sleep(delay)
 msg = self.name + ': num value is ' + str(num)
 print(msg)
 mutex.release()

if __name__ == '__main__':
 for i in range(5):
```

```
delay = 10-2*i
t = MyThread()
t.start()
```

否则，不锁定每个线程，如果第一个线程耗时太长，第二个线程已经修改了数据，那个第一个线程使用的数据就是第二个线程修改过的，而不是第一个线程开始执行时候的期望数值，导致数据顺序错乱。

```
import threading
import time

num = 0
delay = 5
mutex = threading.Lock()

class MyThread(threading.Thread):
 def run(self):
 global num
 global delay

 num = num + 1
 print("now the num is: %d", num)
 time.sleep(delay)
 msg = self.name + ': num value is ' + str(num)
 print(msg)

if __name__ == '__main__':
 for i in range(5):
 delay = 10-2*i # 每个线程延时越来越短，导致后面的线程会先执行
 t = MyThread()
 t.start()

=====output=====
('now the num is:', 1)
('now the num is:', 2)
('now the num is:', 3)
('now the num is:', 4)
('now the num is:', 5)
Thread-5: num value is 5
Thread-4: num value is 5
Thread-3: num value is 5
Thread-2: num value is 5
Thread-1: num value is 5
```

## 2.5 可重入锁（递归锁）

为了满足在同一线程中多次请求同一资源的需求，Python 提供了可重入锁（RLock）。  
RLock 内部维护着一个 Lock 和一个 counter 变量，counter 记录了 acquire 的次数，从而使得资源可以被多次 require。直到一个线程所有的 acquire 都被 release，其他的线程才能获得资源。

具体用法如下：

```
#创建 RLock
mutex = threading.RLock()

class MyThread(threading.Thread):
 def run(self):
 if mutex.acquire(1):
 print("thread " + self.name + " get mutex")
 time.sleep(1)
 mutex.acquire()
 mutex.release()
 mutex.release()
```

## 2.6 守护线程

如果希望主线程执行完毕之后，不管子线程是否执行完毕都随着主线程一起结束。我们可以使用 `setDaemon(bool)` 函数，它跟 `join` 函数是相反的。它的作用是设置子线程是否随主线程一起结束，必须在 `start()` 之前调用，默认为 `False`。

## 2.7 定时器

如果需要规定函数在多少秒后执行某个操作，需要用到 `Timer` 类。具体用法如下：

```
from threading import Timer

def show():
 print("Pyhton")

指定一秒钟之后执行 show 函数
t = Timer(1, show)
t.start()
```

# 3 Python 多进程

## 3.1 创建多进程

Python 要进行多进程操作，需要用到 `multprocessing` 库，其中的 `Process` 类跟 `threading` 模块的 `Thread` 类很相似。所以直接看代码熟悉多进程。

- 方法1：直接使用 `Process`，代码如下：

```
from multiprocessing import Process

def show(name):
 print("Process name is " + name)

if __name__ == "__main__":
 proc = Process(target=show, args=('subprocess',))
 proc.start()
 proc.join()
```

- 方法2：继承 `Process` 来自定义进程类，重写 `run` 方法，代码如下：

```
from multiprocessing import Process
import time

class MyProcess(Process):
 def __init__(self, name):
 super(MyProcess, self).__init__()
 self.name = name

 def run(self):
 print('process name : ' + str(self.name))
 time.sleep(1)

if __name__ == '__main__':
 for i in range(3):
 p = MyProcess(i)
 p.start()
 for i in range(3):
 p.join()
```

## 3.2 多进程通信

进程之间不共享数据的。如果进程之间需要进行通信，则要用到 `Queue` 模块 或者 `Pip` 模块 来实现。

- Queue**

`Queue` 是多进程安全的队列，可以实现多进程之间的数据传递。它主要有两个函数，`put` 和 `get`。

`put()` 用以插入数据到队列中，`put` 还有两个可选参数：`blocked` 和 `timeout`。如果 `blocked` 为 `True`（默认值），并且 `timeout` 为正值，该方法会阻塞 `timeout` 指定的时间，直到该队列有剩余的空间。如果超时，会抛出 `Queue.Full` 异常。如果 `blocked` 为 `False`，但该 `Queue` 已满，会立即抛出 `Queue.Full` 异常。

`get()` 可以从队列读取并且删除一个元素。同样，`get` 有两个可选参数：`blocked` 和 `timeout`。如果 `blocked` 为 `True`（默认值），并且 `timeout` 为正值，那么在等待时间内没有取到任何元素，会抛出 `Queue.Empty` 异常。如果 `blocked` 为 `False`，有两种情况存在，如果 `Queue` 有一个值可用，则立即返回该值，否则，如果队列为空，则立即抛出 `Queue.Empty` 异常。

具体用法如下：

```

from multiprocessing import Process, Queue

def put(queue):
 queue.put('Queue 用法')

if __name__ == '__main__':
 queue = Queue()
 pro = Process(target=put, args=(queue,))
 pro.start()
 print(queue.get())
 pro.join()

```

- **Pipe**

Pipe的本质是进程之间的用管道数据传递，而不是数据共享，这和socket有点像。pipe() 返回两个连接对象分别表示管道的两端，每端都有send() 和recv()函数。

如果两个进程试图在同一时间的同一端进行读取和写入那么，这可能会损坏管道中的数据。

具体用法如下：

```

from multiprocessing import Process, Pipe

def show(conn):
 conn.send('Pipe 用法')
 conn.close()

if __name__ == '__main__':
 parent_conn, child_conn = Pipe()
 pro = Process(target=show, args=(child_conn,))
 pro.start()
 print(parent_conn.recv())
 pro.join()

```

### 3.3 进程池

创建多个进程，我们不用傻傻地一个个去创建。我们可以使用 `Pool` 模块来搞定。

`Pool` 常用的方法如下：

| 方法                         | 含义                                      |
|----------------------------|-----------------------------------------|
| <code>apply()</code>       | 同步执行（串行）                                |
| <code>apply_async()</code> | 异步执行（并行）                                |
| <code>terminate()</code>   | 立刻关闭进程池                                 |
| <code>join()</code>        | 主进程等待所有子进程执行完毕。必须在close或terminate()之后使用 |
| <code>close()</code>       | 等待所有进程结束后，才关闭进程池                        |

具体用法见示例代码：

```

from multiprocessing import Pool
def show(num):
 print('num : ' + str(num))

if __name__=="__main__":
 pool = Pool(processes = 3)
 for i in xrange(6):
 # 维持执行的进程总数为processes, 当一个进程执行完毕后会添加新的进程进去
 pool.apply_async(show, args=(i,))
 print('===== apply_async =====')
 pool.close()
 #调用join之前, 先调用close函数, 否则会出错。执行完close后不会有新的进程加入到pool, join函数等待所有子进程结束
 pool.join()

```

## 3.4 选择多线程还是多进程?

在这个问题上, 首先要看下你的程序是属于哪种类型的。一般分为两种 CPU 密集型 和 I/O 密集型。

- CPU 密集型: 程序比较偏重于计算, 需要经常使用 CPU 来运算。例如科学计算的程序, 机器学习的程序等。
- I/O 密集型: 顾名思义就是程序需要频繁进行输入输出操作。爬虫程序就是典型的 I/O 密集型程序。

如果程序是属于 CPU 密集型, 建议使用多进程。而多线程就更适合应用于 I/O 密集型程序。

作者: 猴哥Yuri

链接: <https://www.jianshu.com/p/a69dec87e646>

来源: 简书

简书著作权归作者所有, 任何形式的转载都请联系作者获得授权并注明出处。

## 执行终端/控制台命令

```

import os
import pexpect

python 执行终端/控制台命令并返回
os.system('ping www.baidu.com') #执行成功 返回 0
#注: os.system() 执行完成 会关闭 所以当执行后续命令需要依赖前面的命令时, 请将多条命令写到一个 os.system() 内

ping = os.popen('ping www.baidu.com').read().strip() #返回输出结果

执行终端命令并交互
ch = pexpect.spawn('命令')
ch.expect('Password:')
ch.sendline('密码')

```

# 检测键盘输入

<https://stackoverflow.com/questions/2408560/python-nonblocking-console-input>

```
#!/usr/bin/env python
...
A Python class implementing KBHIT, the standard keyboard-interrupt poller.
Works transparently on Windows and Posix (Linux, Mac OS X). Doesn't work
with IDLE.

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

...
import os

Windows
if os.name == 'nt':
 import msvcrt

Posix (Linux, OS X)
else:
 import sys
 import termios
 import atexit
 from select import select

class KBHit:

 def __init__(self):
 '''Creates a KBHit object that you can call to do various keyboard things.
 '''

 if os.name == 'nt':
 pass

 else:

 # Save the terminal settings
 self.fd = sys.stdin.fileno()
 self.new_term = termios.tcgetattr(self.fd)
 self.old_term = termios.tcgetattr(self.fd)
```

```

New terminal setting unbuffered
self.new_term[3] = (self.new_term[3] & ~termios.ICANON & ~termios.ECHO)
termios.tcsetattr(self.fd, termios.TCSAFLUSH, self.new_term)

Support normal-terminal reset at exit
atexit.register(self.set_normal_term)

def set_normal_term(self):
 ''' Resets to normal terminal. On Windows this is a no-op.
 '''
 if os.name == 'nt':
 pass
 else:
 termios.tcsetattr(self.fd, termios.TCSAFLUSH, self.old_term)

def getch(self):
 ''' Returns a keyboard character after kbhit() has been called.
 Should not be called in the same program as getarrow().
 '''
 s = ''
 if os.name == 'nt':
 return msvcrt.getch().decode('utf-8')
 else:
 return sys.stdin.read(1)

def getarrow(self):
 ''' Returns an arrow-key code after kbhit() has been called. Codes are
 0 : up
 1 : right
 2 : down
 3 : left
 Should not be called in the same program as getch().
 '''
 if os.name == 'nt':
 msvcrt.getch() # skip 0xE0
 c = msvcrt.getch()
 vals = [72, 77, 80, 75]
 else:
 c = sys.stdin.read(3)[2]
 vals = [65, 67, 66, 68]

 return vals.index(ord(c.decode('utf-8'))))

```

```

def kbhit(self):
 ''' Returns True if keyboard character was hit, False otherwise.
 '''
 if os.name == 'nt':
 return msvcrt.kbhit()

 else:
 dr,dw,de = select([sys.stdin], [], [], 0)
 return dr != []

Test
if __name__ == "__main__":
 kb = KBHit()

 print('Hit any key, or ESC to exit')

 while True:
 if kb.kbhit():
 c = kb.getch()
 if ord(c) == 27: # ESC
 break
 print(c)

 kb.set_normal_term()

```

## 根据键盘输入执行对应任务

There are some sample code about how to remote control [CamJam's fabulous Robot Kit](#) including a piece of code that would allow you to [control a robot with a bluetooth keyboard](#) or the like. Following the example provided below, it should be quite easy to adjust this for whichever keyboard presses you need to detect and then add in the functions you want to trigger from each key press!

```

#!/usr/bin/python3

adapted from https://github.com/recantha/EduKit3-RC-Keyboard/blob/master/rc_keyboard.py

import sys, termios, tty, os, time

def getch():
 # 获取标准输入的描述符
 fd = sys.stdin.fileno()
 # 获取标准输入(终端)的设置
 old_settings = termios.tcgetattr(fd)
 try:
 tty.setraw(sys.stdin.fileno())
 ch = sys.stdin.read(1)

```

```

finally:
 # 使设置生效
 termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
return ch

button_delay = 0.2

while True:
 char = getch()

 if (char == "p"):
 print("Stop!")
 exit(0)

 if (char == "a"):
 print("Left pressed")
 time.sleep(button_delay)

 elif (char == "d"):
 print("Right pressed")
 time.sleep(button_delay)

 elif (char == "w"):
 print("Up pressed")
 time.sleep(button_delay)

 elif (char == "s"):
 print("Down pressed")
 time.sleep(button_delay)

 elif (char == "1"):
 print("Number 1 pressed")
 time.sleep(button_delay)

```

**Note:** It seems to stop in the getch function if no key is pressed which pauses the game. There should be an output if no key is pressed, or a break rather than waiting for a key.

This is because sys.stdin.read() blocks. One option for non-blocking input (at least on Linux) is as follows.

```

#!/usr/bin/python3

import sys, termios, tty, os, time
added
import fcntl

def getch():
 fd = sys.stdin.fileno()
 old_settings = termios.tcgetattr(fd)
 try:
 tty.setraw(sys.stdin.fileno())
 ch = sys.stdin.read(1)

 finally:

```

```

 termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
 return ch

button_delay = 0.2

added
fd = sys.stdin.fileno()
fl = fcntl.fcntl(fd, fcntl.F_GETFL)
fcntl.fcntl(fd, fcntl.F_SETFL, fl | os.O_NONBLOCK)

while True:
 char = getch()

 if (char == "p"):
 print("Stop!")
 exit(0)
 # added
 elif (not char):
 print("No char")
 elif (char == "a"):
 print("Left pressed")
 time.sleep(button_delay)

 elif (char == "d"):
 print("Right pressed")
 time.sleep(button_delay)

 elif (char == "w"):
 print("Up pressed")
 time.sleep(button_delay)

 elif (char == "s"):
 print("Down pressed")
 time.sleep(button_delay)

 elif (char == "1"):
 print("Number 1 pressed")
 time.sleep(button_delay)

```

处理 箭头：

<https://stackoverflow.com/questions/22397289/finding-the-values-of-the-arrow-keys-in-python-why-are-they-ri>

<https://www.raspberrypi.org/forums/viewtopic.php?t=42608>

```

import curses

get the curses screen window
screen = curses.initscr()

```

```

turn off input echoing
curses.noecho()

respond to keys immediately (don't wait for enter)
curses.cbreak()

map arrow keys to special values
screen.keypad(True)

if __name__ == "__main__":
 try:
 while True:
 char = screen.getch()
 if char == ord('q'):
 break
 elif char == curses.KEY_RIGHT:
 # print doesn't work with curses, use addstr instead
 print("right")
 screen.addstr(0, 0, 'right')
 elif char == curses.KEY_LEFT:
 screen.addstr(0, 0, 'left ')
 print("left")
 elif char == curses.KEY_UP:
 screen.addstr(0, 0, 'up ')
 print("up")
 elif char == curses.KEY_DOWN:
 screen.addstr(0, 0, 'down ')
 print("down")
 finally:
 # shut down cleanly
 curses.nocbreak(); screen.keypad(0); curses.echo()
 curses.endwin()

```

## 按任意键退出

初学Python时在总想实现一个按任意键继续/退出的程序(受.bat毒害), 奈何一直写不出来, 最近学习Unix C时发现可以通过 `termios.h` 库来实现, 尝试一下发现Python也有这个库, 所以终于写出一个这样的程序. 下面是代码:

```

#!/usr/bin/env python
-*- coding:utf-8 -*-
import os
import sys
import termios

def press_any_key_exit(msg):
 # 获取标准输入的描述符
 fd = sys.stdin.fileno()

 # 获取标准输入(终端)的设置
 old_ttyinfo = termios.tcgetattr(fd)

 # 配置终端

```

```

new_ttyinfo = old_ttyinfo[:]

使用非规范模式(索引3是c_lflag 也就是本地模式)
new_ttyinfo[3] &= ~termios.ICANON
关闭回显(输入不会被显示)
new_ttyinfo[3] &= ~termios.ECHO

输出信息
sys.stdout.write(msg)
sys.stdout.flush()
使设置生效
termios.tcsetattr(fd, termios.TCSANOW, new_ttyinfo)
从终端读取
os.read(fd, 7)

还原终端设置
termios.tcsetattr(fd, termios.TCSANOW, old_ttyinfo)

if __name__ == "__main__":
 press_any_key_exit("按任意键继续...")
 press_any_key_exit("按任意键退出...")

```

其他关于 `termios` 的信息可以参考Linux手册:

`man 3 termios`

另补充一下\*nix终端的三种模式(摘自<Unix-Linux编程实践教程>)

## 规范模式

规范模式, 也被成为cooked模式, 是用户常见的模式.驱动程序输入的字符保存在缓冲区, 并且仅在接收到回车键时才将这些缓冲的字符发送到程序.缓冲数据使驱动程序可以实现最基本的编辑功能, 被指派这些功能的特定键在驱动程序里设置, 可以通过命令stty或系统调用tcsetattr来修改

## 非规范模式

当缓冲和编辑功能被关闭时, 连接被成为非规范模式.终端处理器仍旧进行特定的字符处理, 例如处理Ctrl-C及换行符之间的转换, 但是编辑键将没有意义, 因此相应的输入被视为常规的数据输入 程序需要自己实现编辑功能

## raw模式

当所有处理都被关闭后, 驱动程序将输入直接传递给程序, 连接被成为raw模式.

### 相关资源:

1. <https://stackoverflow.com/questions/13207678/whats-the-simplest-way-of-detecting-keyboard-input-in-python-from-the-terminal>
2. <https://raspberrypi.stackexchange.com/questions/28302/python-script-with-loop-that-detects-keyboard-input>
3. <http://www.zmonster.me/2015/03/02/keyboard-control-with-python.html>
4. <https://www.jianshu.com/p/03010ac70e4c>

5. <https://blog.csdn.net/howard2005/article/details/79446980>
6. <https://codeday.me/bug/20181024/320687.html>

这些答案都不适合我。这个包，pynput，正是我所需要的。<https://pypi.python.org/pypi/pynput>

```
from pynput.keyboard import Key, Listener
def on_press(key):
 print('{0} pressed'.format(
 key))
def on_release(key):
 print('{0} release'.format(
 key))
 if key == Key.esc:
 # Stop listener
 return False
Collect events until released
with Listener(
 on_press=on_press,
 on_release=on_release) as listener:
 listener.join()
```

## 数据分析

### 基本概念

统计学中的基本概念：

- 统计学里最基本的概念就是样本的均值、方差、标准差。首先，我们给定一个含有n个样本的集合，下面给出这些概念的公式描述：
- 均值：

$$\bar{X} = \frac{\sum_{i=1}^n X_i}{n}$$

- 标准差：

$$S = \sqrt{\frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1}}$$

- 方差：

$$S^2 = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1}$$

- 均值描述的是样本集合的中间点，它告诉我们的信息是有限的，而标准差给我们描述的是样本集合的各个样本点到均值的距离之平均。

以这两个集合为例，[0, 8, 12, 20]和[8, 9, 11, 12]，两个集合的均值都是10，但显然两个集合的差别是很大的，计算两者的标准差，前者是8.3后者是1.8，显然后者较为集中，故其标准差小一些，标准差描述的就是这种“散布度”。

之所以除以n-1而不是n，是因为这样能使我们以较小的样本集更好地逼近总体的标准差，即统计上所谓的“无偏估计”。详见<https://www.zhihu.com/question/20099757>（马同学）而方差则仅仅是标准差的平方。

### 为什么需要协方差：

**标准差和方差一般是用来描述一维数据的**，但现实生活中我们常常会遇到含有多维数据的数据集，最简单的是大家上学时免不了要统计多个学科的考试成绩。面对这样的数据集，我们当然可以按照每一维独立的计算其方差，但是通常我们还想了解更多，比如，一个男孩子的猥琐程度跟他受女孩子的欢迎程度是否存在一些联系。**协方差就是这样一种用来度量两个随机变量关系的统计量**，我们可以仿照方差的定义：

$$\text{var}(X) = \frac{\sum_{i=1}^n (X_i - \bar{X})(X_i - \bar{X})}{n-1}$$

来度量各个维度偏离其均值的程度，协方差可以这样来定义：

$$\text{cov}(X, Y) = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{n-1}$$

协方差的结果有什么意义呢？如果结果为正值，则说明两者是正相关的（从协方差可以引出“相关系数”的定义），也就是说一个人越猥琐越受女孩欢迎。如果结果为负值，就说明两者是负相关，越猥琐女孩子越讨厌。如果为0，则两者之间没有关系，猥琐不猥琐和女孩子喜不喜欢之间没有关联，就是统计上说的“相互独立”。

从协方差的定义上我们也可以看出一些显而易见的性质，如：

$$1、\text{cov}(X, X) = \text{var}(X)$$

$$2、\text{cov}(X, Y) = \text{cov}(Y, X)$$

### 皮尔森相关系数(Pearson correlation coefficient)

$$p_{x,y} = \text{corr}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_x \sigma_y} = \frac{E[(X - u_x)(Y - u_y)]}{\sigma_x \sigma_y}$$

$$p_{x,y} = \text{corr}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_x \sigma_y} = \frac{E[(X - u_x)(Y - u_y)]}{\sigma_x \sigma_y} \quad (1)$$

- 协方差就是俩人跳舞的舞步协同程度，如果一起向前走或者向后退，协方差就是正值；如果一个朝前一个朝后，协方差就是负值；如果各自都不动，就是零。
- 相关系数就是标准化的协方差，就是剔除了俩人舞步尺度大小不一的影响。
- 协方差对于相关系数 等价于 标准差对于变异系数。（这里感谢 x2yline 的指点！）

## 协方差矩阵

前面提到的猥琐和受欢迎的问题是典型的二维问题，而协方差也只能处理二维问题，那维数多了自然就需要计算多个协方差，比如n维的数据集就需要计算

$$\frac{n!}{(n-2)! * 2}$$

个协方差，那自然而然我们会想到使用矩阵来组织这些数据。给出协方差矩阵的定义：

$$C_{n \times n} = (c_{i,j}, \quad c_{i,j} = \text{cov}(\text{Dim}_i, \text{Dim}_j))$$

这个定义还是很容易理解的，我们可以举一个三维的例子，假设数据集有三个维度，则协方差矩阵为：

$$C = \begin{pmatrix} \text{cov}(x,x) & \text{cov}(x,y) & \text{cov}(x,z) \\ \text{cov}(y,x) & \text{cov}(y,y) & \text{cov}(y,z) \\ \text{cov}(z,x) & \text{cov}(z,y) & \text{cov}(z,z) \end{pmatrix}$$

可见，协方差矩阵是一个对称的矩阵，而且对角线是各个维度的方差。

## 相关系数矩阵图

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = np.random.rand(1000, 5)
df = pd.DataFrame(data) # 将numpy数组转为df
correlations = df.corr() # 计算变量之间的相关系数矩阵

plot correlation matrix
fig, ax = plt.subplots(figsize=(9,9)) #调用figure创建一个绘图对象

cax = ax.matshow(correlations, vmin=-1, vmax=1) #绘制热力图，从-1到1

fig.colorbar(cax) #将matshow生成热力图设置为颜色渐变条
ticks = np.arange(0,5,1) #生成0-9，步长为1
ax.set_xticks(ticks) #生成刻度
ax.set_yticks(ticks)
#ax.set_xticklabels(names) #生成x轴标签
#ax.set_yticklabels(names)
plt.show()
```

## 热力图（相关系数矩阵图）

```
import seaborn as sns
import numpy as np
import pandas as pd
a = np.random.rand(4,3)
fig, ax = plt.subplots(figsize = (9,9))
#二维的数组的热力图，横轴和数轴的ticklabels要加上去的话，既可以通过将array转换成有column
#和index的DataFrame直接绘图生成，也可以后续再加上去。后面加上去的话，更灵活，包括可设置labels大小方向等。
sns.heatmap(pd.DataFrame(np.round(a,2), columns = ['a', 'b', 'c'], index = range(1,5)),
 annot=True, vmax=1,vmin = 0, xticklabels= True, yticklabels= True,
 square=True, cmap="YlGnBu")
#sns.heatmap(np.round(a,2), annot=True, vmax=1,vmin = 0, xticklabels= True, yticklabels=
True,
square=True, cmap="YlGnBu")
ax.set_title('二维数组热力图', fontsize = 18)
ax.set_ylabel('数字', fontsize = 18)
ax.set_xlabel('字母', fontsize = 18) #横变成y轴，跟矩阵原始的布局情况是一样的
```

```
ax.set_yticklabels(['一', '二', '三'], fontsize = 18, rotation = 360,
horizontalalignment='right')
ax.set_xticklabels(['a', 'b', 'c'], fontsize = 18, horizontalalignment='right')
```

<https://zhuanlan.zhihu.com/p/35494575>

## 数据结构

### 队列

先进先出队列(简称队列)是一种基于先进先出(FIFO)策略的集合类型.

队列的最简单的例子是我们平时碰到的,比如排队等待电影，在杂货店的收营台等待，在自助餐厅排队等待（这样我们可以弹出托盘栈）。行为良好的线或队列是有限制的，因为它只有一条路，只有一条出路。不能插队，也不能离开。你只有等待了一定的时间才能到前面。下图展示了一个简单的 Python 对象队列。



队列是有序数据集合，队列的特点，删除数据项是在头部，称为前端(front)，增加数据在尾部，称为后端(rear)

## Queue

```
导入队列
#from queue import Queue
from multiprocessing import Queue

最多接收3个数据
q = Queue(3)

put 向队列中添加数据
q.put(1)
q.put(2)
q.put(3)

获取当前队列长度
print(q.qsize())

取出最前面的一个数据 1 , 还剩两个
print(q.get())

再加入数据
q.put(4)

#超过三个了.如果没有timeout参数会处于阻塞状态,卡在那边.若设置2秒,2秒后会raise 一个 FULL的报错
q.put(5, timeout=2)

当然,也可以直接给个 block=False,强制设置为不阻塞(默认为会阻塞的),一旦超出队列长度,立即抛出异常
q.put(6, block=False)

同样的,当取值(get)的次数大于队列的长度的时候就会产生阻塞,设置超时时间为最多等待x秒,队列中再没有数据,就抛出异常.也可以使用block参数,跟上面一样
```

其他常用方法:

```
empty: 检查队列是否为空, 为空返回True, 不为空返回False
full : 判断队列是否已经满了

join & task_done :
q = Queue(2)
q.put('a')
```

```

q.put('b')
程序会一直卡在下面这一行，只要队列中还有值，程序就不会退出
q.join()

q = Queue(2)
q.put('a')
q.put('b')

q.get()
q.get()
插入两个元素之后再取出两个元素，执行后发现，程序还是卡在下面的那个join代码
q.join()

q = Queue(2)
q.put('a')
q.put('b')

q.get()
get取完队列中的一个值后，使用task_done方法告诉队列，我已经取出一个值并处理完毕，下同
q.task_done()
q.get()
在每次get取值之后，还需要在跟队列声明一下，我已经取出了数据并处理完毕，这样执行到join代码的时候才不会被卡住
q.task_done()
q.join()

```

## 双向队列

但是删除列表的第一个元素（抑或是在第一个元素之前添加一个元素）之类的操作是很耗时的，因为这些操作会牵扯到移动列表里的所有元素。

### deque

`collections.deque` 类（双向队列）是一个线程安全、可以快速从两端添加或者删除元素的数据类型。而且如果想要有一种数据类型来存放“最近用到的几个元素”，`deque` 也是一个很好的选择。这是因为在新建一个双向队列的时候，你可以指定这个队列的大小，如果这个队列满员了，还可以从反向端删除过期的元素，然后在尾端添加新的元素。使用示例如下：

```

>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)

```

```
>>> dq.extendleft([10, 20, 30, 40]) ❸
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

- ❶ maxlen 是一个可选参数，代表这个队列可以容纳的元素的数量，而且一旦设定，这个属性就不能修改了。
- ❷ 队列的旋转操作接受一个参数 n，当 n > 0 时，队列的最右边的 n 个元素会被移动到队列的左边。当 n < 0 时，最左边的 n 个元素会被移动到右边。
- ❸ 当试图对一个已满 (`len(d) == d maxlen`) 的队列做尾部添加操作的时候，它头部的元素会被删除掉。注意在下一行里，元素 0 被删除了。
- ❹ 在尾部添加 3 个元素的操作会挤掉 -1、1 和 2。
- ❺ `extendleft(iter)` 方法会把迭代器里的元素逐个添加到双向队列的左边，因此迭代器里的元素会逆序出现在队列里。

## 其他队列

<https://my.oschina.net/yangyanxing/blog/296052>

Python提供的所有队列类型：

1. 先进先出队列 `queue.Queue`
2. 后进先出队列 `queue.LifoQueue` (`Queue`的基础上进行的封装)
3. 优先级队列 `queue.PriorityQueue` (`Queue`的基础上进行的封装)
4. 双向队列 `queue.deque`

除了上述提到的队列与双端队列,还有两个用的比较少的:后进先出队列与优先级队列

## 自己队列实现

在实际编码中不会自己来实现一个队列.因为python本身就有自带的队列库.如果想自己实现可以利用列表的一些特性,比如.append或者.pop来实现.也可以抛开列表重新定义一个队列.这里有一个很好的例子来实现

<http://zhaochj.github.io/2016/05/15/2016-05-15-%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84-%E5%8D%95%E7%AB%AF%E9%98%9F%E5%88%97/>

## 相关链接

<https://facert.gitbooks.io/python-data-structure-cn/3.%E5%9F%BA%E6%9C%AC%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/3.10.%E4%BB%80%E4%B9%88%E6%98%AF%E9%98%9F%E5%88%97/>  
<https://docs.lvrui.io/2016/07/20/Python%E4%B8%AD%E5%85%88%E8%BF%9B%E5%85%88%E5%87%BA%E9%98%9F%E5%88%97queue%E7%9A%84%E5%9F%BA%E6%9C%AC%E4%BD%BF%E7%94%A8/>

## 其他

### @ 修饰符

在模块或者类的定义层内对函数进行修饰。出现在函数定义的前一行，不允许和函数定义在同一行。

```
def funcA(A):
 print("function A")
 print(A)

def funcB(B):
 print(B(2))
 print("function B")

@funcA
@funcB
def func(c):
 print("function C")
 return c**2
```

--output--

function C

4

function B

function A

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x

x