

## 本文档版本修订记录

时间	版本	修订主要内容	修订者
2019. 3. 1	初稿 v1. 0		fuyihang

# Python 环境搭建

---

## Python 简介

Python 作为一门解释性的程序开发语言(脚本语言)，因其简单易学，而且功能强大。在 2019 年 3 月份，编程语言流行指数 PYPL 排行榜中，Python 已经超越 Java 成为最受欢迎的开发语言。

Python 在下列几大主流应用领域都广受好评：

- 网络编程，Web 应用开发
- 图形界面开发（GUI 开发）
- 游戏开发
- 科学计算
- 网络爬虫
- 数据分析
- 机器学习、人工智能
- 系统运维
- 自动化测试

## 核心程序安装

安装 Python 核心程序有两种方式：纯净安装和打包安装。

### 纯净安装

纯净安装，指的是直接安装官方提供的 Python 核心程序（包括解释器、核心程序、标准模块库等）。然后再根据自己的需求，下载并安装其它的扩展库。

由于 Python 中涉及到大量扩展库，以及扩展库的不同的版本。如果自己安装的话，**缺点是：**需要自行考虑各种扩展库之间的版本依赖关系，比较麻烦；**好处是：**可以根据自己的需要来有选择性地安装各种扩展库。

### 打包安装

最常见的打包安装，指的就是下载 Anaconda 安装包进行安装。

Anaconda 是 Python 的科学计算发行版，不但已经包含了 Python 核心程序，而且还包含众多的流行的科学、数学、工程、数据分析的 Python 包等等。**好处是：**这样可以免去自行安装众多扩展库的麻烦，而且 Anaconda 还会考虑各种扩展库的版本依赖性，不用担心各扩展库间的版本的兼容性。

推荐第 2 种安装方式，简单。

直接去 <https://www.anaconda.com/distribution/#download-section> 下载相应操作系统（Windows/macOS/Linux）的 Python 安装包，直接安装即可。在此不再赘述。

随 Python 包安装的还有一批应用程序：

Anaconda Navigator，用于管理环境和包

Anaconda Prompt，使用命令行界面来管理环境和包

Spyder, 面向科学开发的 IDE

Jupyter Notebook, 使用命令行界面来进行交互编程

## 运行 Python 程序三种方式

安装 Python 后, 就可以开始使用 Python 来进行程序开发了。

运行 Python 有三种方式: 一是采用交互式运行 Python 语句, 二是命令方式运行 Python 文件, 三是脚本方式运行 Python 文件。

## 交互式运行

进入 Python 交互界面, 逐条输入 Python 语句执行。

如果安装的 Anaconda 发布版, 则直接运行 Anaconda Prompt。

(注: 如果是在 macOS 系统中, 由于允许 Python 多版本存在, 要使用最新版本, 需要使用 python3 命令)

详细的命令原型:

```
python [-bBdEhiIOqsSuvVWx?] [-c command | -m module-name | script | - ] [args]
```

请参考 <https://docs.python.org/3/using/cmdline.html>

- 查看 Python 安装位置

```
$which python3
```

```
/Users/fusx/anaconda3/bin/python3
```

- 查看 python 版本

```
$python -version
```

```
Python 3.7.2
```

- 进入交互式命令行界面

```
$python
```

Python 3.7.2 (default, Dec 24 2018, 02:44:43) [MSC v.1915 64bit (AMD64)] :: Anaconda, Inc. on win32

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
```

看到交互提示符>>>, 就表示已经进入 Python 交互界面。

## ● 输入 python 语句

我们使用 python 的第一条语句, 通常如下:

```
>>>import this
```

这条命令会输出 “Python 之禅”, 即优秀代码的指导原则。然后, 就可以逐条输入 python 语句。

## • 打印出字符串 (第一条语句)

```
>>>print("Hello Python World!")
```

## ➤ 退出 python 交互界面

要退出交互界面, 可直接按 Ctrl+D(maxOS 中), Ctrl+Z(Windows 中), 也可以在命令行中直接输入 exit() 或 quit() 退出。

```
>>>exit()
```

就回到操作系统的命令行。

## 命令运行文件

将 Python 语句写在一个 .py 文件中, 一次性执行文件中所有语句。

假定 python 语句编写在一个文件中, 比如 hello.py 文件, 现在可以一次性运行该脚本文件。

打开终端界面, 输入

```
>python hello.py
```

如果脚本文件需要带参数，则参数跟在文件后面

```
>python hello.py 参数1 参数2
```

（当然，如果要处理输入的参数，则在 hello.py 脚本中需要增加代码处理）

## 脚本执行文件

把 Python 文件（假定是 hello.py）当成普通的脚本执行。相当于，将 Python 程序发布后，直接运行 Python 程序。

步骤如下：

- 1) 首先在.py 文件的首行加入特殊的 shebang 行（以#!开头）

```
#!/Users/fusx/anaconda3/bin/python3
```

（其中，上述路径可以使用 `which python3` 得到）

- 2) 然后，设置脚本文件的执行权限。

```
sudo chmod a+x hello.py
```

- 3) 最后，就可以在命令行直接运行脚本文件了。此时，就不会输入 python 命令了。

```
./hello.py
```

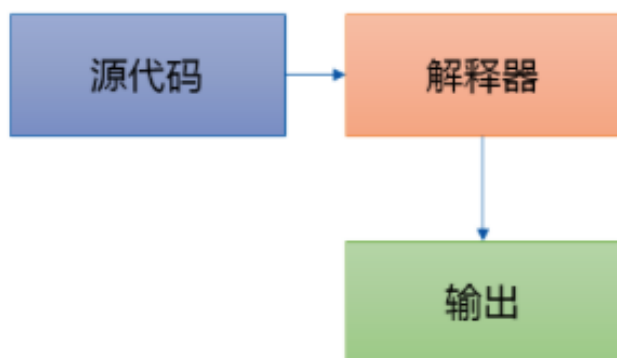
把 Python 文件当成脚本执行，除了第一步，其余都是系统本身通用的配置操作。

## Python 解释器

### 解释器种类

Python 是一门解释器语言，所有的代码都必须通过解释器执行。

Python 解释器有多种实现方式，有用 C 实现的，有用 Java 实现的。



## 解释器：边解释边执行

比较常见的有：

### 1) CPython(Cython)

这是官方默认的 Python 解释器，用 C 语言写的。当执行 Python 代码时，CPython 会将代码转化成字节码(bytecode)。

CPython 用 `>>>` 作为提示符。默认情况下，你进入的是 CPython 解释器的交互界面。

### 2) IPython

基于 CPython 之上的一个加强版交互式解释器，执行代码时和 CPython 完全一样。但支持语法高亮，支持变量、模块、函数、类的自动补全，支持 shell 命令，内置许多特殊功能函数，比如 `%pwd` 显示当前目录，`%env` 显示环境变量等。

IPython 用 `In [序号]:` 作为提示符。后面我们使用的 Jupyter Notebook 使用的就是 IPython 解释器。

### 3) PyPy

这个是用 Python 语言本身写成的解释器，PyPy 会把代码转化成机器码，所以可显著提高 Python 代码的执行速度。

不过，同样的代码在 PyPy 执行和在 CPython 执行，其执行结果会有不同。

### 4) Jython

用 Java 实现的解释器。Jython 允许把 Java 的模块加载到 python 模块中使用。Jython 使用了 JIT 技术，运行程序时会先转化成 Java 字节码，使用使用 JRE 执行；程序还可以把 Python 代码打包成 jar 包。

## 5) IronPython

用 C#实现的解释器，可以用在微软.NET 平台上，直接将 Python 代码编译成.Net 字节码。

## 开发工具选择

要进行 Python 程序开发，就要用到开发工具，开发工具基本分为两大类：

### 一类是采用交互式开发

交互式开发，相当于直接在解释器中进行交互，逐条输入 Python 语句，马上输出运行结果。常见的就是直接用 CPython 进行交互，或者使用更高级一点的交互工具 Jupyter Notebook 进行开发。这一类适合于教学、调试和小型程序开发等。

### 二类是使用 IDE 开发

对于复杂的大型的 Python 项目，就需要使用集成开发环境（IDE）了。常见的 IDE 有：PyCharm、VSCode、Sublime Text 等等。

PyCharm：据说是最好的 Python 开发工具，不过需要付费。

VSCode：微软最好的开发工具，同样功能强大，关键还免费。

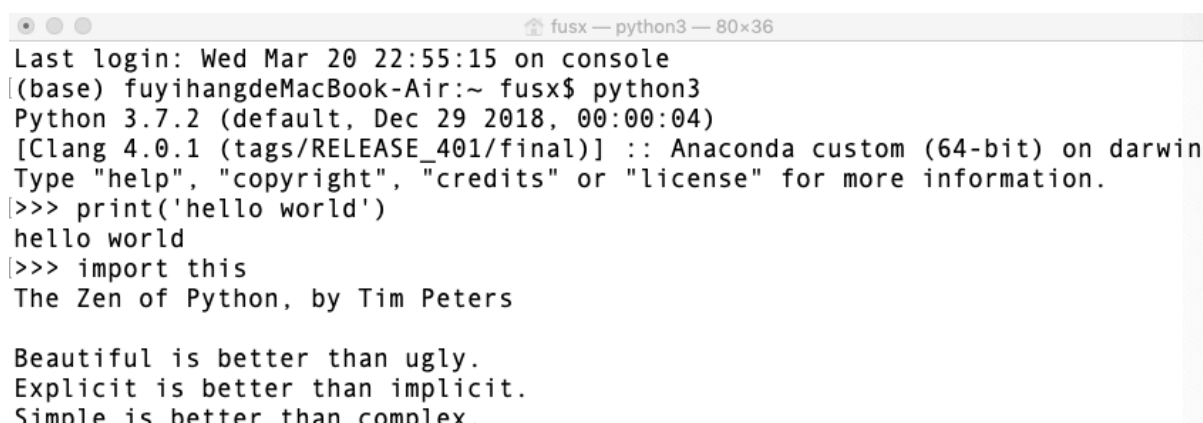
Sublime Text：开源的工具，时不时弹出打赏界面，有点烦。

## 1) 直接用 CPython 交互

默认，就安装了官方的 CPython 解释器，可以直接使用。

一般地，当你在命令行中输入 python3 启动 Python 时，使用的就是官方默认的 CPython 解释器和交互界面。但是，官方的

CPython 解释器在输入多行语句和复杂语句时不太方便。所以，大多数开发者会选择功能更强大的解释器。



```
fusx — python3 — 80x36
Last login: Wed Mar 20 22:55:15 on console
(base) fuyihangdeMacBook-Air:~ fusx$ python3
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda custom (64-bit) on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world')
hello world
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

## 2) 用 Jupyter Notebook 交互

较多的开发者，选择 Jupyter Notebook 开发工具和环境，它是基于 IPython 解释器的一个 GUI 交互开发界面，基于 Web 浏览器的开发环境，方便项目文件的管理。它支持注释/代码混写，支持多行代码编写等等。

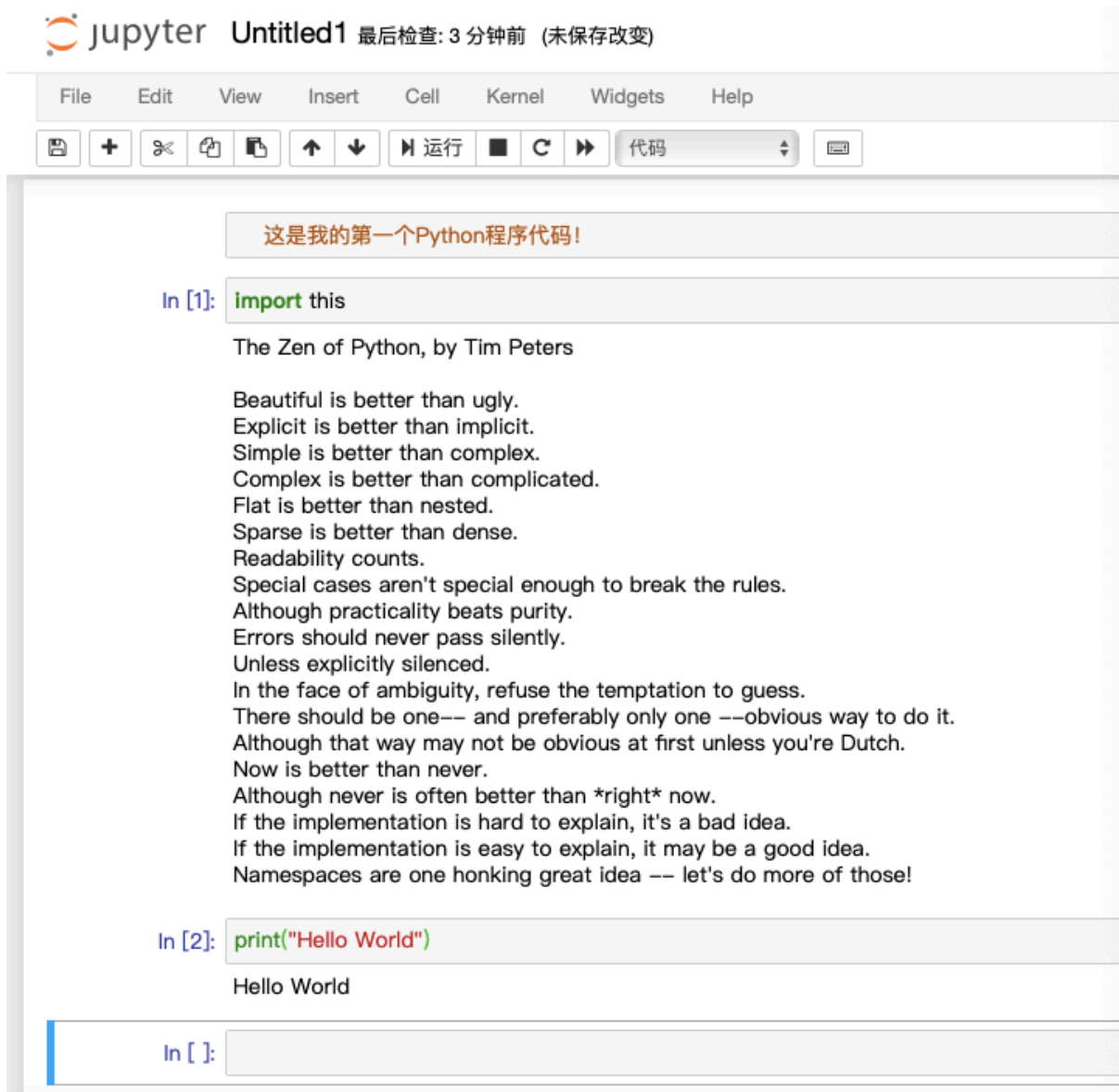
一般在教学中，经常使用这个交互式工具。即能够注释，同时也能够执行 Python 代码。

如果你在前面安装了 Anaconda，随之安装的还有 IPython 解释器，以及 Jupyter Notebook 交互工具。你可以使用 `pip show ipython` 来查看一下 ipython 的安装信息。

现在，运行 Anaconda Navigator，你可以看到 jupyter Notebook，点击运行（Launch），然后就会打开一个 Web 浏览器，点击右上角的 New->Python3 就会建一个 Untitled1 文件，于是就进入 Jupyter Notebook 交互界面。

你可以输入各种 Python 语句，进行交互，并且可以保存脚本，点击 Untitled1，还可以重命名脚本文件。





### 3) 使用集成 IDE 开发

当然，如果涉及到大型的 Python 项目，肯定是不大会直接在交互工具中进行开发的。此时，就需要用到集成开发环境（IDE）了，如 Pycharm，VSCode 等。后续会告诉大家如何安装这些 Python 代码编辑器，如下为 VSCode IDE 的界面。



后续复杂的项目，我们将会使用 VSCode 来开发。现在，我们将先使用交互工具来开发。输入任何一条 Python 语句，能够立马看到输出结果。

## Python 包管理

Python 的功能之所以强大，就在于 Python 中提供了大量的可扩展的包，这些包可以根据需要进行随时安装和使用。Python 中提供了两种方式对包进行管理：pip 和 conda。

建议在安装 Python 之后，升级所有的包（参考下面的命令）

## 包管理工具

### pip 工具

pip 是常用的 python 自带的包管理工具，可以用来很方便地安装和管理各种三方库。类似于 java 的 maven，RedHat 中的 yum。

当前 pip 最新版本是 19.0。如果你安装了多个 Python 版本，则有可能你需要输入 pip3 -version 来尝试。

>pip install -upgrade pip

表示更新 pip 自己到最新版本

>pip -h

查看帮助

>pip -version	检查是否安装了 pip 工具
>pip list	查看安装了哪些包。
>pip list --outdated	查看哪些包需要更新
>pip show --files Somepackage	查看已安装的包
>pip install Somepackage	安装指定的包
>pip install --upgrade Sompacage	升级包
>pip uninstall Sompacage	卸载包

## conda 工具

如果安装了 Anaconda, 则提供了 conda 包管理工具。打开 Anaconda Prompt 运行如下命令。

> conda list	查看已经安装的包
> conda search search_term	搜索安装包
> conda install Somepackage	安装包, 会自动安装依赖项。
> conda remove Somepackage	卸载包
> conda update Somepackage	更新指定包
> conda upgrade --all	更新所有的包, 提示是否更新时输入 y(Yes)

## 工具差异

pip 与 conda 命令的区别如下:

作用	pip基本命令	conda基本命令
查看版本	<code>pip --version</code>	<code>conda --version</code>
升级版本	<code>pip install -U pip</code>	<code>conda update conda</code>
列出所有安装包	<code>pip list</code>	<code>conda list</code>
检测更新	<code>pip list --outdated</code>	N
更新某个库	<code>pip install --upgrade #库名字</code>	<code>conda update #库名字</code>
安装(更新)某个库为指定版本	<code>pip install "#库名字==版本号"</code>	<code>conda install #库名字=版本号</code>
更新所有库	N	<code>conda update --all</code>
卸载库	<code>pip uninstall #库名字</code>	N

相比之下，pip 命令不如 conda 命令强，conda 命令可以一条命令将所有待更新的包都升级 `conda update -all`。

建议在刚安装好 Python 后，在终端命令行中输入 `conda update -all` 以便全部更新所有的包和模块。

## 包安装方式

安装 Python 包和库一般有两种方式：一种在线安装，另一种是离线安装。

### 在线安装

在线安装，指的是在联网的情况下，直接使用前面的 pip/conda 命令安装想要的包或库。此时，电脑会自行下载相应的包并安装在本地硬盘上。

```
$pip install numpy #安装 numpy 包
```

### 离线安装

离线安装，指的是已经将待安装的包或库下载到本地硬盘（一般扩展名为 .whl），此时不需要电脑联网，就可以本地安装。

```
$pip install numpy-1.15.4-cp37-none-win_amd64.whl
```

下载包和库的官方网址: <https://pypi.org/>

## 虚拟环境管理

当前, 由于 Python 的版本在不断升级, 而且 Python3 并不完全是向后兼容的, Python2 中的一些语句在 Python3 中不能够直接使用。典型的比如打印 print 语句, 在 Python2 中是一条 print 语句, 但在 Python3 中却是一个函数 print()。

```
>>>print "hello world"          #Python2 中这样使用
>>>print("hello world")         #Python3 中这样使用
```

这会导致在 Python 升级后影响部分 Python 程序的运行。为了解决这个问题, Python 引入了虚拟环境(virtualenv)这个概念, 即允许 Python 的多个版本共存, 而且不同的 Python 程序可以在不同的虚拟环境中运行, 互不影响。

所以, 虚拟环境是用来隔离 Python 项目/程序的, 不同的虚拟环境中可以有不同版本的 Python 库。

下面以 conda 命令为例, 来说明虚拟环境相关的常用命令。

## 查看可用环境

```
$conda env list
```

列出你创建的所有可用的虚拟环境名称。名称前带有一个星号(\*)的, 表示是当前正在使用的虚拟环境。

一般, Python 有一个默认的虚拟环境叫 base。

## 创建环境

```
$conda create -n py2
```

这样就在路径/Users/username/anaconda3/envs/py2 下创建了一个虚拟环境，名为 py2，不过默认的 Python 版本是 Python3。

如果想创建指定的 Python 2.7 版本虚拟环境，则可以使用下列命令：

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

## 进入环境

虚拟环境创建后，就可以进入虚拟环境，或叫激活虚拟环境。

请使用命令 `conda activate py2`，进入环境后，就会在提示符中看到带括号的环境名称比如 `(py2)~$`。

进入命令后你就可以使用 `pip` 或 `conda` 来安装所需要的特定版本的扩展库。

## 导出环境

现在，你已经在你的虚拟环境中安装了特定版本的包或库。你可以共享你的虚拟环境，以便其他人安装并使用你的 Python 程序。因此，你可以导出你的环境中所用到的库的版本。

```
$ conda env export -f mpy2env.txt
```

这样就将环境中所用到的所有包的名称，以及版本都写入了该文件。

## 离开环境

完成开发后，如果想离开当前的虚拟环境，则可以使用命令 `conda deactivate`，就离开当前环境，进入到默认的虚拟环境。

## 共享环境

现在，你可以将你导出的环境文件交给其他人，其他人可以创建你的环境。

```
$conda env create -f mypy2env.txt
```

这样其他人就创建了一个新环境，而它包含了文件中列出的同样版本的包或扩展库。此时，其他人就可以正常地运行你的程序。

## 删除环境

如果你不想使用自建的环境了，那么先离开此环境，然后就可以删除此环境。

```
$conda env remove -n py2
```

## 代码编辑器安装

要进行大型的 Python 项目开发，需要借助于开发工具。最常见的 Python 代码编辑器有 Pycharm, VSCode, sublime Text, Emacs 等等。你可以根据习惯选择自己熟悉的 IDE。

理论上，Pycharm 是最出色的 IDE，但是这个工具是收费的；其次是微软的 Visual Studio Code，这是免费的。

其实，前面当你安装 anaconda 时，也会有一个选项，让你安装微软的 VSCode。

## 安装 VSCode

在 <https://code.visualstudio.com/download> 下载安装包，然后双击 VSCodeUserSetup-x64-1.30.1.exe 进行安装即可。

## 配置 VSCode

微软的 VSCode 可以支持多种开发语言，所以，在使用 VSCode 开发前需要作一些基本的配置。

### 配置解释器

要使用 VSCode 开发 Python 代码，必须要先配置解释器。

打开 VSCode，按 `ctrl+shift+P`，搜索 `select interpreter`，选择 `python3` 解释器即可。

### 安装 *Python* 扩展

打开 VSCode，打开左边框最后一个扩展按钮，搜索 `python extension`，然后安装此扩展包即可。

### 配置中文界面

如果你觉得英文界面不好用，你还可以配置为中文界面（可选）。

打开 VSCode，打开左边框最后一个扩展按钮，搜索 `chinese`，选择简体中文，点击右侧的 `Install` 安装。

用 `Ctrl+Shift+P` 打开配置选项【`configure display language`】，打开 `local.json` 文件。将“`en`”改为“`zh-cn`”（注：输入冒号会自动显示可选择语言），保存即可。

现在重启 VSCode，大功告成！

## Python 开发语言基础

---

要快速地掌握 Python 开发语言基础，建议首先阅读官方的文档：



#官方参考文档（英文）：

<https://docs.python.org/3/index.html>

#官方参考文档（中文）：

<https://docs.python.org/zh-cn/3.7/index.html>

#Python 基础课程（中文文档）：

<http://www.runoob.com/python3/python3-file-methods.html>

还有其它中文参考文档：

#Python 中文开发者社区

<https://www.pythontab.com>

#腾讯 Python 开发者手册

<https://cloud.tencent.com/developer/doc/1198>

## 一、基本格式

### 编码

默认情况，Python3 以 UTF-8 编码，所有字符串都是 unicode 字符串。如果是其它的编码，可以在第一行或者第二行说明编码情况。

一般情况下，Python 代码文件的第一第二行如下所示：

```
#!/usr/local/bin/python3
# -*- coding: utf-8 -*-
```

第一行是 UNIX 的“shebang 行”，以#!开头，表示 python 脚本是可以被直接执行。当然，你还得在 Unix 中设置可执行的权限，比如使用命令 `chmod a+x hello.py`（其中 `hello.py` 表示是你的 python 脚本文件）。

常见的第一行如下所示，还可以用来指明 python 的版本或虚拟环境等。

```
#!/usr/bin/python      #默认的 python

#!/usr/bin/python2.7   #指定的 python 版本

#!/usr/bin/env python3 #会搜索执行路径（虚拟环境）启动 python

#!/usr/bin/python -v   #还可带选项
```

第二行，声明文件所使用的编码，形如：

```
# -*- coding: encoding -*-
```

其中 encoding 可以是 Python 支持的任意一种编码，默认就是 utf-8 编码。

当然，这两行在 Python 程序中写不写都无所谓，因为都是注释行，所以在 Python 程序中不会执行，这两行只是在操作系统环境中会作专门的解释。

## 注释

#单行注释用#开头

#多行注释可以用三个单引号'''，'或三个双引号"""包围。

```
#第一行注释
'''
print( "...")    #这一块代码全部被注释掉
'''

"""
print( "...")    #这一块代码全部被注释掉
"""
```

## 缩进

大多数的开发语言，缩进是为了代码易读，不同的代码块采用大括号 {} 括起来。但在 Python 中，缩进有特别的含义，缩进表示的是同一个逻辑的代码块。

缩进的空格数是可变的，但同一个代码块的必须包含相同的缩进空格数。一般情况下，每次缩进，默认是 4 个空格（一个 Tab 符）。

```
>>> if Ture:
    print(“对”)
    #其它处理
else:
    print(‘错’)
```

## 标识符

标识符，指的是 Python 中的变量名、函数名、类名等。

标识符的命名，必须遵守如下规则：

1. 由字母、数字和下划线组成，第一个字符必须是字母或下划线
2. 标识符区分大小写（即大小写敏感）。
3. 标识符不能包括空格，不能用关键字作为标识符（请参考后续关键字小节）
4. 标识符也应该尽量避免使用内置的标识符（参考后面内置的标识符），以免造成很多种冲突，出现意想不到的后果。
5. 一般情况下，标识符的命名的写法遵循两种模式：
  - a) 驼峰命名法：即包含多个单词时首字母大写，其余字母小写，比如 `userName`，或 `UserName`。
  - b) 下划线分隔法：即包含多个单词时用下划线隔开，比如 `user_name`，`sum_list`。
6. 在 Python 中，注意标识符的大小写约定，以便识别：

- a) 对于模块名，一般是全部小写字母；
- b) 对于内置的标识符(函数、类)，都全部是小写字母；
- c) 对于自定义类名，一般首字母大写（驼峰式）；
- d) 对于类成员，一般首字母小写（驼峰式或下划线分隔）
- e) 对于变量名，一般首字母小写（驼峰式或下划线分隔）；
- f) 对于自定义常量，一般全部字母大写（保留关键字除外）。

## 保留关键字 *keyword*

Python 中定义有保留关键字，这些关键字是不能用作自定义标识符，可以用下列命令查看保留字。

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

## 内置标识符

Python 中有一些内置的标识符（包括内置常量、内置函数等）。虽然，理论上自定义的标识符是可以与这些内置标识符同名的，但是，在程序中应该尽量避免与这些名称重名，否则容易引起混淆。

#列出有哪些内置的函数和变量

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

列出的标识符包括：内置常量、内置异常、内置函数、标准数据类型名称，及一些特定的名称。列出的标识符包括如下几类：

## 1) 内置特殊对象

- True, False 这是 bool 类的两个常量。
- None 是 NoneType 类唯一的常量对象，表示没有值
- Ellipsis 是 ellipsis 类的常量对象，与扩展切片语法结合使用的特殊值。
- NotImplemented 是 NotImplementedType 类的常量对象，是 `__eq__()`, `__lt__()` 可以返回的特殊值，表示没有针对其他类型实施
- quit, exit, copyright, credits, license 是由 site 模块导入的常量，对交互式外壳有用，不应该在程序中使用

## 2) 内置函数

内置函数也可直接引用，不需要明确导入，详细可参考链接：

<https://docs.python.org/3/library/functions.html>。

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

### 3) 内置标准数据类型

类似于 bool, int, str, list, tuple, dict, set, frozenset 等。

### 4) 内置的操作运算符

象 and, or, not 布尔运算符。更多的操作运算符请参考后续章节。

### 5) 内置异常类

异常类也是内置的，不需要明确导入。详细请参考后续章节或链接：

<https://docs.python.org/zh-cn/3.7/library/exceptions.html>

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        +-- BufferError
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- EnvironmentError
            +-- IOError
            +-- OSError
                +-- WindowsError (Windows)
                +-- VMSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            +-- NotImplementedError
        +-- SyntaxError
            +-- IndentationError
            +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            +-- UnicodeError
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError
    +-- Warning
```

```
+++ DeprecationWarning
+++ PendingDeprecationWarning
+++ RuntimeWarning
+++ SyntaxWarning
+++ UserWarning
+++ FutureWarning
+++ ImportWarning
+++ UnicodeWarning
+++ BytesWarning
```

## 6) 系统特殊属性

比如 `__build_class__`, `__debug__`, `__doc__`, `__package__`, `__spec__`

`__loader__`: 模块的调用者

`__name__`: 对象的名称

`__dict__`: 存储对象（可写）属性的字典或其他映射对象

`__class__`: 类对象所属的类

`__base__`: 类对象的父类

`__bases__`: 类对象的父类的元组

`__name__`: 所有对象实例的名称

## 7) 系统内置类

所有模块类 `module`

内置函数类 `builtin_function_or_method`

内置类方法 `method_descriptor`

自定义类方法 `method`

自定义类的类 `type`

所有自定义类的基类 `object`

其它常用类 `bool`, `NoneType`, `ellipsis`, `NotImplementedType`

## 书写格式

### 一条语句在一行

一般情况下，一个物理行是 80 个字符，通常一行写完一条 Python 语句，即一条逻辑语句书写在一个物理行。

```
>>>msg=' 第一条消息！'
>>>print(msg)
```

### 一条语句在多行

但如果语句太长，可以使用反斜杠(\)来实现多行语句，这样容易阅读和理解。

```
if 1900 < year < 2100 and 1 <= month <= 12 \
    and 1 <= day <= 31 and 0 <= hour < 24 \
    and 0 <= minute < 60 and 0 <= second < 60:
    return 1
```

### 一行有多条语句

当然，Python 也可以在同一行中使用多条语句，语句之间使用分号(;)分割。一般不建议采用这种写法。

```
>>>msg=' 第一条消息！' ; print(msg)
```

## 二、五大基本语句

Python 中有很多简单语句，也有很多复合语句。下面我们将会重点介绍常用的五种语句（赋值、输入输出、条件判断、循环、异常）。



但是，在讲基本语句之前，有必要先理解两个基本概念：变量和对象。因为几乎所有的语句都会涉及到对变量或对象的操作。

## Python 变量

变量，是对数据对象的一个引用。

Python 是弱类型开发语言，相比其它强类型开发语言（比如 C++，Java 语言等），Python 变量有所不同。

- 1) Python 变量是不需要事先定义或声明的。
- 2) 变量在使用前必须赋值，赋值后，相当于变量指向了内存中的某个对象。
- 3) Python 变量是没有类型的，可以随时赋值为其它类型（不象 C++ 语言，声明为整型后，是不可能赋值为字符串；但在 Python 却允许给变量赋予任何类型的数值）。

```
#变量是没有类型的，但对象是有类型
>>>n = 3
>>>type(n)          #n 为整型 int
<class 'int' >

>>>n = 'OK'          #n 为字符串型 str
>>>type(n)
<class 'str' >
```

### 1.赋值语句

赋值语句是任何编程语言的基础，指的就是给变量赋值，或称变量实例化。变量赋值，其实就是给一个数据对象一个引用的名字，或者说，一个变量指向了内存中的一个数据对象。

## 变量赋值

赋值语句的格式：**变量名 赋值运算符 数据**。

最典型的赋值运算符就是等号（=），即初始化变量和赋值。

- 1、 可以给单个变量赋值。
- 2、 也可以同时为多个变量赋值。
- 3、 赋值语句还有更高级的用法，可以用来分解复杂的序列对象。

#1. 基本赋值语句

```
name = 'Jack'
age = 20
```

#2. 多变量赋值

```
a = b = c = 2
```

#3. 多变量赋不同的值

```
a, b, c = 10, 'Hello', 2+3j
```

#=====高级用法=====

#使用赋值语句来分解复杂数据

#4. 实现序列对象的分解

```
data = [ 'abc', 3, (2012, 12, 3) ]          #列表
name, num, dt = data
print(name, num, dt)    #现在 name=' abc' , num=3, dt=(2012, 12, 3)
```

#5. 任意长度对象分解

```
data = ( 'adc', '01028673684', '13876282543' )
name, *phoneNum = data
print(name)          #打印机 'adc'
print(phoneNum) #打印出[ '01028673684', '13876282543' ]
```

## 赋值运算符

除了等号，还有其它赋值运算符。比如+=，-=等。

为了避免重复，详细见运算符小节。

```
n = 1

n += 3      #此时 n = n+3 =4

n = 2      #此时，n = n*2 = 8
```

## 2.1 输出语句 print

在 Python2 中，print 只是一条标准输出语句，但是在 Python3 中，print 是一个内置函数。按照常规，我们依然把它当成一条语句。在 Python 中，输入第一条打印语句往往如下打印出” Hello, World”：

```
print(“Hello, World!”)      #输出一个字符串
print(“Hello”, ” world”)    #输出多个字符串
```

### *print* 语句原型

```
print(*objects, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

print 函数参数说明如下：

- 1) \*objects，说明 print 函数的实参个数可以是任意数量的。
- 2) sep 用于分隔多个字符串，默认为空格字符串。
- 3) end 用于输出字符串后接着输出的字符串，默认为换行符\n。
- 4) file 表示输出的位置，默认是标准输出窗口 sys.stdout。
- 5) flush 表示输出时是否可缓存。

## 基本输出语句

- 1) 可以输出单个字符串，也可以输出多个字符串。
- 2) 多个字符串之间，默认为空格隔开，也可以自定义分隔符。
- 3) 输出字符参数后，默认再输出一个换行符\n，也可以根据需要进行修改。

```
x = 'a'
y = 'b'

#1. 换行输出
print(x)
print(y)

#2. 不换行输出
x='a'
y='b'
print(x,end=' ')      #输出后紧跟一个空格
print(y,end=' ')
print('OK')           #此处有换行

#3. 多个字符串之间，用/隔开
print(2019, 2, 3, sep=' /' )

-----以上代码输入结果为-----
a
b
a b OK
2019/2/3
```

## 带变量输出

如果输出的字符串中，有部分取值是由变量指定，则需要格式化字符串。格式化字符串有如下方式：

- 1) 采用字符前缀 F 或 f，其中嵌入参数，参数用 {} 括起来。
- 2) 采用字符串类 str.format 函数，用大括号 {} 表示输出格式，括号及其里面的字符将会被变量参数替换。
- 3) 使用 % 操作符，这种格式已经过时，不建议使用。

```

>>>name = '小明'
>>>age = 20

#1. 字符前缀
>>>msg = f'客户名是{name}, 年龄是{age}'
>>>print(msg)

#2. str.format() 函数
msg = '客户名是 {}, 年龄是 {}'.format(name, age)

#过时的表示, 不建议使用
>>>print('我叫%s 名, 今年%d 岁。'%( '小明', 10))
我叫小明名, 今年 10 岁。

```

在使用 str.format 格式化字符串时, 如果有多个变量要输出, 则要注意位置对齐。

- 4) 如果括号中没有数字, 则参数要严格按位置顺序对应;
- 5) 括号中也可以指定数字, 则数字用于指向参数的位置顺序。
- 6) 如果括号中指定关键字参数, 则按照关键字匹配。
- 7) 特别地, 对于字典变量类型, 输出格式需要带[键名]。

```

>>>name = '小明'
>>>age = 20

#1. 程序会按顺序位置依次匹配变量
>>>print('客户名是 {}, 年龄是 {} 岁。'.format(name, age))

#2. {N}N 指定变量的位置顺序
>>>print('客户名是 {1}, 年龄是 {0} 岁。'.format(age, name))

#3. 使用关键字参数, 变量位置无关
>>>print('客户名是 {name}, 年龄是 {age} 岁。'.format(age=25, name='张三'))

#4.1 使用字典, 格式形如 {0[key]}
>>>table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>>print('Jack: {0[Jack]}; Sjoerd: {0[Sjoerd]}; '
      'Dcab: {0[Dcab]}'.format(table))

```

#4.2 使用字典内容，带\*\*标记，表示使用键名

```
>>>print(' Jack: {Jack}; Sjoerd: {Sjoerd}; Dcab: {Dcab}'.format(**table))
```

## 输出彩色字体

在高级应用中，print 也可以在控制台打印出了带颜色的字符串。

要给字符串带有颜色，则格式如下：

`\033[显示方式;字体色;背景色 m + 字符串 + \033[0m`

显示方式、前景色、背景色是可选参数，由于表示三个参数的值都是唯一的，所以参数可以没有先后顺序。

字体色	背景色	颜色描述
30	40	黑色
31	41	红色
32	42	绿色
33	43	黄色
34	44	蓝色
35	45	紫红色
36	46	青蓝色
37	47	白色

显示方式	效果
0	终端默认设置
1	高亮显示
4	下划线
5	闪烁
7	反白显示
8	不可见
22	非粗体
24	非下划线
25	非闪烁
27	非反显

```
>>>msg = '\033[31m' + '我爱中国' + '\033[0m' #红色字体
```

```
>>>print(msg)
```

我爱中国

```
>>>print('\033[0;33m 我爱中国\033[0m']          #正常，黄色字体，无背景

>>>print('\033[1;31;46m 我爱中国\033[0m']          #高亮，红色字体，青色背景

>>>print('\033[5;33m 我爱中国\033[0m']          #闪烁，黄色字体

print('\033[0;36m 床前明月光，')
print('疑是地上霜。')
print('举头望明月，')
print('低头思故乡。 \033[0m']
```

## 2.2 输入语句 Input

input 语句，用于程序输入，函数会一直等待用户输入，按回车后函数结束等待，并返回用户输入的字符串（不包括回车换行符）。

原型：input([prompt])

```
msg = input( "请输入一个字符串： " )
print( "你输入的内容是： " , msg)

msg = input( "请输入一个整数： " )
age = int(msg)          #将用户输入强制转换为整数，注意异常处理
```

## 3. 条件判断 if-elif-else

完整的条件判断语句为 if-elif-else 格式，其中只有 if 语句 是必选的，其余是可选的。而且，elif 语句可重复多次，甚至 if 语句可以嵌套另一个 if 语句。

if 语句经常和比较运算符（==, !=, <, <=, >, >=）、布尔运算符、成员运算符、身份运算符等结合使用。

(注: Python 中没有 switch-case 语句)

```
>>>car = 'BMW'
>>>if car == 'BMW':
    print( 'OK' )
OK

>>>car == 'bmw'
False

>>> n =5
>>> if n >= 3:
    print( 'OK' )
OK

#判断语句
age = input( '请输入你的年龄: ' )
if age <10:
    Print( '你不需要买票入场' )
elif age <15:
    print( '你需要买半票入场' )
elif age <60:
    print( '你需要买全票入场' )
else:
    print( '你不能够入场' )

>>>lt = list( 'abcd' )
>>>' a' in lt
True
>>>' f' in lt
False

#空列表判断
>>>lt == []
False

>>>lt = [1,2]
>>>lt == [1,2]
True
```



## 4.1 遍历循环 for-in-else

常见的循环语句一般有两个：for 语句和 while 语句，其中 for 语句为遍历循环语句，而 while 为条件循环语句。

1. for 循环可以遍历任何序列的项目，如列表或字符串等。
2. break 语句用于跳出当前循环体（跳到循环体外）。
3. continue 语句用于继续循环（跳过循环体内其余的代码块，继续下次循环判断）。
4. else 语句（可选）在穷尽列表后被执行，但循环被 break 终止时不执行。

```
>>>languages = [ 'C' , ' C++' , ' Python' , ' Java' ]
>>>for x in languages:
    print(x)
>>>for i in range(5):          #输出 0~4
    print(i)
    else:
        print(“循环正常结束” )

>>>for letter in 'Runoob' :    #遇到 o 就中止整个循环，不会打印 else 语句
    if letter == 'o' :
        break
    print( '当前字母:' , letter)
    else:
        print( '循环结束' )

>>>for letter in 'Runoob' :    #遇到 o 跳过本次打印，循环完会打印 else 语句
    if letter == 'o' :
        continue
    print( '当前字母:' , letter)
    else
        print( '循环结束' )

>>>for n in range(2,10):
    for x in range(2,n):
        if n%x ==0:            #n 是偶数，终止该 for 循环
            print(n, ' 等于' , x, ' *' , n/x)
            break
        else:
            print(n, ' 是质数' )
```

```
#输出结果
2 是质数
3 是质数
4 等于 2*2
5 是质数
6 等于 2*3
7 是质数
8 等于 2*4
9 等于 3*3
```

## 4.2 条件循环 while-else

1. 条件满足就执行循环内代码，条件不满足则执行 else 代码块，并中止整个循环。
2. 可以使用 Ctrl+C 来中断循环。
3. break、continue 的使用和前面 for 完全一样。
4. 如果遇到 break 语句，else 语句不会被执行；如果是 continue，则 else 语句会被执行。

```
>>>sum = 0
>>>counter = 1
>>>while counter <= 100:      #依次累加 1~100 的数
    sum += counter
    counter+=
>>>print(sum)                 #输出 5050

>>>count = 0
>>>while count < 3:
    print(count, " 小于 3" )
    count += 1
>>>else:
    print(count, " 大于或等于 3" )
>>>print( "循环结束" )
0 小于 3
1 小于 3
2 小于 3
3 大于或等于 3
循环结束

>>> var = 10
>>>while var > 0:              #遇到 5 就中止循环, 而且不打印 else 语句
```

```

        if var == 5:
            break
        print( '当前变量值:' , var)
        var = var -1
    else:
        print( '循环结束' )

>>>var = 10
>>>while var > 0:                #遇到 5 就不打印，其余都要打印，包括 else 语句
    if var == 5:
        var = var -1
        continue
    print( '变量值: ' , var)
    var = var - 1
else:
    print( '循环结束' )

```

## 5.异常语句 try-except-else-finally

对异常的捕获和处理，请参考后面异常处理章节。

## 其它特殊语句

其它特殊语句，可以单独使用，也有可能和其它复杂语句配合使用。

- 1、 assert 语句
- 2、 break, continue 循环中止/跳过语句
- 3、 classdef/fundef 语句，类定义和函数定义语句
- 4、 del 删除语句，用于删除对象或元素
- 5、 eval 语句，用来执行表达式，返回结果
- 6、 future 语句
- 7、 global, nonlocal 语句，变量作用域描述语句
- 8、 import 导入语句，用于导入模块和函数

- 9、 pass 空语句，用做占位语句，不做任何事。
- 10、 raise 语句，抛出异常
- 11、 with 上下文管理语句，常用于文件打开，会自动关闭文件对象
- 12、 yield 语句，实现迭代器，支持遍历

### 三、七类操作运算符

#### 1.赋值运算符(8 个)

运算符	描述
=	简单赋值
+=	加法赋值
-=	减法赋值
*=	乘法赋值
/=	除法赋值
%=	取模赋值
**=	乘幂赋值
//=	取整赋值

a += b，相当于 a = a+b，其余类似

## 2.算术运算符(8 个)

运算符	描述	实例
+	加法	
-	减法	
*	乘法	
/	除法	
%	取余-返回整除的得数的余数	21%10=1 -21%10=9 21%-10=-9
**	幂, 返回x的y次幂 =pow(x, y)	2**3=8
//	向下取整	21//10=2 -21//10=-3
-x	取x的相反数	

## 3.位运算符(6 个)

操作符	含义	实例 a = 0011 1100 = 60 b = 0000 1101 = 13
x&y	按位与	a&b=0000 1100 =12
x y	按位或	a b=0011 1101 =61
x^y	按位非	a^b=0011 0001 =49
~x	按位取反, 相当于-x-1	~a =1100 0011 =-61
x<<n	左移n位, 高位丢弃, 低位补0	a<<2=1111 0000 =240
x>>n	右移n位, 高位补0, 低位丢弃	a>>2=0000 1111 =15

## 4.比较运算符(6 个)

所有比较运算符返回布尔值（1 表示真 True，返回 0 表示假 False）。

操作符	含义
==	是否相等
!=	是否不相等
<	是否小于
<=	小于或等于
>	是否大于
>=	大于或等于

## 5.布尔运算符(3 个)

运算符	描述	实例
and 布尔与	x和y均为True, 就返回True, 否则返回False	x and y
or 布尔或	x和y只要有一个是True, 就返回True, 否则返回False	x or y
not 布尔非	返回x的相反值	not x

注意：

- 1) and 是一个短路操作符，所以如果第一个参数为真，它只会计算第二个参数。
- 2) or 是一个短路运算符，因此如果第一个参数为假，它只会计算第二个参数。

- 3) `not` 具有比非布尔运算符更低的优先级，因此 `not a == b` 被解释为 `not (a == b)`，并且 `a == not b` 是语法错误。

## 6.成员运算符(2 个)

运算符	描述	实例
<code>in</code>	如果值在序列中就返回True，否则返回False	
<code>not in</code>	是in的相反判断	

```
>>>lt = [1, 2, 3]
>>>a = 2
>>>a in lt
True

>>> b =4
>>>b in lt
False
```

## 7.身份运算符(2 个)

用于比较两个对象的存储单元。`is` 与`==`的区别：

- 1) `is` 用于判断两个变量引用对象是否为同一个
- 2) `==`用于判断引用变量的值是否相等。

运算符	描述	实例
is	判断两个对象的id是否引用自同一个对象，即对象id是否相等	x is y, 等同于 id(x)==id(y) 如果相同，则返回True, 否则返回False
is not	是is的相反判断	等同于id(x) != id(y)

```

>>>a = [1, 2, 3]
>>>b = a
>>> b is a
True
>>> b == a
True

>>>b = a[:]          #b 为 a 的复制

>>> b == a           #值内容相等
True
>>> b is a           #但不是同一个对象
False

```

## 运算符优先级

下表列出从高到低的优先级运算符：

运算符	描述
() , [] , {}	括号、元组、列表、字典、集合等
x[index], x(), x.attribute	下标、切片、函数/属性引用
await x	await 表达式
**	指数，幂
+x, -x, ~x	正号、负号、位反
*, /, //, %, @	算术运算符和@



<code>+, -</code>	加、减
<code>&lt;&lt;, &gt;&gt;</code>	位移
<code>&amp;</code>	按位与
<code>^</code>	按位取反
<code> </code>	按位或
<code>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, !=, ==</code>	比较/成员/身份运算符
<code>not x</code>	布尔非
<code>and</code>	布尔与
<code>or</code>	布尔或
<code>if-else</code>	条件表达式
<code>lambda</code>	匿名函数表达式

#### 四、六种数据类型

python 中提供了大量可用的标准数据类型，标准数据类型大致可以分为两大类：

- 1) 一类是内置的数据类型，在 Python 核心程序中，即不用导入其它模块，可在程序中直接使用；
- 2) 另一类是其它标准的数据类型，随着 Python 自动安装，不需要自行下载安装的，但这些标准数据类型一般在某个模块中，在使用前需要使用 import 导入的。

内置的数据类型有 6 个（或 6 类）：

1. Number（数字）
2. String（字符串）
3. Tuple（元组）

4. List（列表）

5. Set（集合）

6. Dictionary（字典）

其中，前 3 个是不可变数据类型（不能修改其中部分数据），后 3 个是可变数据（可以修改其中的数据）。

## Python 对象

### 对象四要素

在 Python 中，一切皆对象（这句话在后面有更深入的理解，此处略过上万字）。

对象，可以看成是内存中的一个真实存在的数据，是有内存指针的。

任何一个 Python 对象，通常有四要素：id, type, value 和 alias。Id 是对象的唯一标志，type 表示这个对象的数据类型，value 表示这个对象的具体的值，alias 是这个对象的引用名称。

#### 1) 对象 id

- a) 对象 id 在创建后从来不会改变，相当于内存中的地址，每个对象都有唯一的 id。
- b) Id() 函数可以返回对象 id（内存指针），因此可以用来判断两个变量指向的对象是不是一样的。

#### 2) 对象 type

- a) 相当于变量或对象保存的数据类型。
- b) 可用内置函数 type() 来查看对象的数据类型。

#### 3) 对象 value

- a) 对象 value 就是对象真实的值。
- b) 一般情况下，可用内置函数 print() 直接输出对象的值。

#### 4) 对象 alias（可选要素）

- a) 对象 alias（别名）就是变量名。变量名其实就是对象的引用名称。
- b) 有时候对象是可以没有别名的（比如用完就丢弃，不需要再次引用）。
- c) 所谓的实例化, 即在内存中创建了一个真实的对象；而所谓变量赋值，其实就是变量指向了内存中一个真实的数据对象。此时，使用变量和直接使用对象是完全一样的，所以多数时候，我们都是用变量来代替数据引用，以做到简洁易读。

```
#对象 id
>>> id(3)
4467530992
>>> id(4)
4467531024
>>> id(5)
4467531056
>>>
>>> n = 3
>>> id(n)
4467530992

>>> n2 = n
>>> id(n) == id(n2)
True

#对象 type
>>> type(n)
<class 'int'>

>>> type('OK')
<class 'str'>

#对象 value
>>> print(n)
3
```

## 对象类型识别

由于 Python 变量是没有类型的，可以指向任何数据类型的对象，要想知道当前变量保存的是什么样的数据类型，或者两个变量指向的是否是同一个对象，经常会涉及到如下三个内置函数：

- 4) `is` 身份运算符，用来判断两个对象是否指向的是同一个对象。其实 `is` 操作符就是比较的是两个对象的 ID 是否一样，如果一样表示这两个对象是同一个。
- 5) `type()` 函数，用来打印出对象的数据类型。
- 6) `isinstance()` 函数，来判断变量保存的对象是不是指定的数据类型。

注：`ininstace` 和 `type` 的区别在于：`type()` 不会认为子类是一种父类；而 `isinstance` 会认为子类也是一种父类类型。

#is 身份操作符

```
>>>n = 3
```

```
>>>n2 = 3
```

```
>>> n is n2
```

```
True
```

```
>>>id(n) == id(n2)
```

#n2 变量也指向同一个对象

#用 `is` 操作符检验这两个对象是不是同一个

```

True

>>> n3 = 5
>>> n3 is n
False
>>> n3 is not n
True

#类型 type()
>>>a, b, c, d = 10, 5.5, True, 5+3j
>>>print(type(a), type(b), type(c), type(d))      #查询变量类型
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>

#对象类型判断 isinstance
>>>isinstance(a, int)
True

```

## 变量与对象的指向

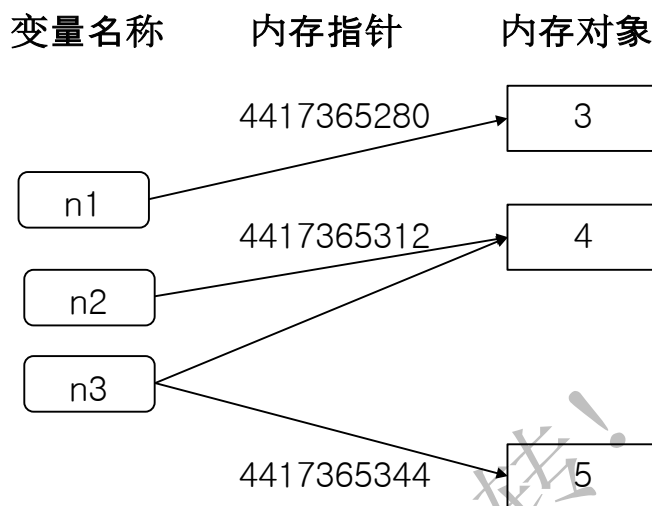
如下代码所示：

第 1 行代码 `id(3)`，此时 Python 在内存中创建一个对象 3；  
 第 2 行代码 `id(4)`，此时 Python 在内存中创建一个对象 4；  
 ——可知这两个对象的 `id` 是不一样的（`id` 号相差 32，知道为何？）  
 第 3 行代码 `n1=3`，给对象 3 一个名字（变量名）；  
 第 4-5 行，可知 `n1` 和 3 其实指的是同一个对象；  
 第 6-7 行，相当于给对象 4 取了两个别名 `n2, n3`；  
 第 8-9 行，可知 `n2、n3、4` 这三个描述的是同一个对象；  
 第 10 行，`n3=5` 是赋值语句，你认为会发生什么？会不会将内存中的对象 4 修改为 5 了？不会，因为整数是不可变数据类型，是不能修改的。所以，Python 会重新在内存中创建一个对象 5，将 `n3` 指向对象 5；  
 第 11-12 行，表明 `n2` 和 `n3` 已经是不一样的对象了。

```

>>> id(3)
4417365280
>>> id(4)
4417365312
>>> n1 = 3
>>> id(n1) == id(3)
True
>>> n1 is 3
True
>>> n2 = 4
>>> n3 = 4
>>> id(n2) == id(n3)
True
>>> id(n3)
4417365312
>>> n2 is n3
True
>>> n3 = 5
>>> id(n2) == id(n3)
False
>>> id(n3)
4417365344
>>> 

```



## 变量指向变化

变量修改（变量赋值）时，变量的指向是否会发生变化，是会受到变量的数据类型的影响的。

1) 右边例子中使用的是 int（不可变数据类型），原来 n2 和 n 指向同一个对象 2，当修改 n2 时，n 不会发生变化，说明此时 n2 指向和 n 指向不同了。

2) 左边例子中使用的是 list（可变数据类型），原来 lt2 和 lt 指向同一个列表对象，现在修改 lt2，但是 lt 也发生了改变，说明此时 lt2 指向和 lt 指向还是相同的。

所以总结一下：

- 不可变类型变量修改，会发生变量指向改变；
- 可变类型变量修改，不会发生变量指向改变。

```

>>> lt = [1,2,3]
>>> lt2 = lt
>>> lt2 is lt
True
>>> lt2[0] = 5
>>> lt
[5, 2, 3]
>>> lt2 is lt
True
>>>

[>>> n = 2
>>> n2 = n
>>> n2 is n
True
>>> n2 = 3
>>> n
2
>>> n2 is n
False
>>> ]

```

## 1. 数字类型 Number

<https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

### 四种数字类型

Python 中数字有四种类型:

1. int 整数, 长整型
2. bool 布尔, 如 True, False (True 为 1, False 为 0)
3. float 浮点数, 如 1.23、4.1E-2
4. complex 复数, 如 1+2j、1.2+3.4j

```

n = 2
n = 0xA0F          #十六进制 n=2575
n = 0o37           #八进制 n=31
n = 1+2j

```

## 运算符(算术和位运算)

数字类型的对象可以进行两种类型的运算，算术运算符和位运算符，请参考前面章节的操作运算符。

## 数字格式化输出

正常情况下，`print` 语句会正确地识别变量类型，并输出相应的格式。但如果想进一步指定输出格式，可以采用格式化符号来指定。

- 1) 通过在参数后面接冒号(:)和格式符，可按指定的格式输出。
- 2) 也可以增加格式辅助符，进一步指定格式。

### 常用格式符

下面是常用的数字型格式化。

字符	含义	Notes
<code>d</code>	格式化整数	
<code>b</code>	格式化二进制	
<code>o</code>	格式化八进制	(1)
<code>x</code>	格式化十六进制（小写）	(2)
<code>X</code>	格式化十六进制（大写）	(2)
<code>e</code>	浮点指数形式（小写）-科学计数法	(3)
<code>E</code>	浮点指数形式（大写）-科学计数法	(3)



字符	含义	Notes
f	格式化浮点十进制数字	(3)
F	格式化浮点十进制数字	(3)
'g'	%f 和 %e 的简写 Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
c	格式化字符（可以是整数或单个字符）	
!a	字符中(用 <code>ascii()</code> 转换)	
!s	字符串(用 <code>str()</code> 转换)	
!r	字符串(用 <code>repr()</code> 转换)。好像不支持了??	

```
#各种进制表示
msg = "int: {0:d},
hex: {0:x}, oct: {0:o}, bin: {0:b}, float: {0:f}, Exponent: {0:e}, char: {0:c}".format
(42)
msg2 = "str: {0:s}".format('OK')
print(msg, msg2)
```

## 格式辅助符

格式辅助符是对格式符的进一步的补充描述。

Flag	Meaning
'#'	The value conversion will use the “alternate form” (where defined below). 添加前缀。在八进制数前面显示('0o'), 在十六进制前面显示'0x' 或者 '0X' (取决于用的是'x' 还是'X'), 二进制数前面显示 0b
'0'	The conversion will be zero padded for numeric values. 数字前面用 0 来填充, 而不是空格
'_'	The converted value is left adjusted (overrides the '0' conversion if both are given). 左对齐。正常情况下, 字符串默认是左对齐, 数值默认是右对齐。
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+' or '-'	A sign character ('+' or '-') will precede the conversion (overrides a “space” flag). 在正数前面显示加号( + ). 默认负数前面是有负号的, 但正数前面是没有正号的。
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)
*	数字, 定义输出的宽度
%	输出百分数

#指定小数尾数

```
>>>import math
```

```
>>>print( '常量 PI 的值为{0:.3f}。' .format(math.pi)) #返回 PI 值 3.142
```

```
>>>">{0:5d}" .format(42)
```

```
'    42'
```

```
>>>">{0:+5d}" .format(42)
```

```
'   +42'
```

```

>>> "{0:#o}".format(42)
'0o52'
>>> "{0:+#o}".format(42)
'+0o52'
>>> "{0:+#x}".format(42)
'+0x2a'
>>> "{0:+#b}".format(42)
'+0b101010'
>>> "{0:+#b}".format(-42)
'-0b101010'
>>> "{0:%}".format(0.1523)
'15.230000%'
>>> "{0:.3%}".format(1.23456)
'123.456%'
>>> "{0:10.2%}".format(0.1523)
'      15.23%'

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f' {name:10} ==> {phone:10d}')
...
Sjoerd      ==>      4127
Jack        ==>      4098
Dcab        ==>      7678

#!a 表示用 asii(), !s 表示用 str(), !r 表示用 repr()
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```



## 常用数值和数学库

```

# numbers      #数值基类
# math         #数学函数
# cmath        #数学函数+复数
# decimal      #浮点数
# fractions    #分数

```

```
# random          #随机数
# statistics      #统计
```

## 常量数字

需要导入 math 库。

常量	描述(需要导入 math 库)
pi	数学常量 pi (圆周率, 一般以 $\pi$ 来表示)
e	数学常量 e, e 即自然常数 (自然常数)。

## 数学函数

需要导入 math 库。

函数	返回值 ( 描述 )
<u>abs(x)</u>	绝对值, 如 abs(-10) 返回 10
<u>ceil(x)</u>	向上取整数, 如 math.ceil(4.1) 返回 5
<u>sqrt(x)</u>	返回数字 x 的平方根。
<u>exp(x)</u>	返回 e 的 x 次幂( $e^x$ ), 如 math.exp(1) 返回 2.718281828459045
<u>fabs(x)</u>	返回数字的绝对值, 如 math.fabs(-10) 返回 10.0
<u>floor(x)</u>	返回数字的下舍整数, 如 math.floor(4.9) 返回 4
<u>log(x)</u>	如 math.log(math.e) 返回 1.0, math.log(100, 10) 返回 2.0
<u>log10(x)</u>	返回以 10 为基数的 x 的对数, 如 math.log10(100) 返回 2.0

<u><code>max(x1, x2, ...)</code></u>	返回给定参数的最大值，参数可以为序列。
<u><code>min(x1, x2, ...)</code></u>	返回给定参数的最小值，参数可以为序列。
<u><code>modf(x)</code></u>	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。
<u><code>pow(x, y)</code></u>	<code>x**y</code> 运算后的值。
<u><code>round(x [, n])</code></u>	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。

## 随机数函数

需要导入 random 库

函数	描述
<code>choice(seq)</code>	从序列的元素中随机挑选一个元素，比如 <code>random.choice(range(10))</code> ，从 0 到 9 中随机挑选一个整数。
<code>randrange([start,] stop [, step])</code>	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
<code>random()</code>	随机生成下一个实数，它在 [0, 1) 范围内。
<code>seed([x])</code>	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。

<code>shuffle(lst)</code>	将序列的所有元素随机排序
<code>uniform(x, y)</code>	随机生成下一个实数，它在 $[x, y]$ 范围内。

## 三角函数

需要导入 `math` 库。

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

## 2. 字符串 String

<https://docs.python.org/3/library/string.html>

## 字符串定义

1. 字符串可以用单引号'和双引号"定义
2. 使用三引号('''或''')可以指定一个多行字符串。
3. 系统会自动将两个字符串（中间没有其它字符）拼接。
4. 字符串可以用 + 运算符连接在一起。
5. 字符串用 \* 运算符表示重复。
6. Python 没有单独的字符类型，一个字符就是长度为 1 的字符。

```
msg = 'Hello World! '           #用'表示
msg = "第一个消息"             #用"表示
msg = '这个字符包含"'         #混用

#用"""表示多行字符串
paragraph = """这是多行字符串。
其中可以使用制表符
TAB(\t)。
也可以使用换行符[\n]。
"""

print(paragraph)

#如果某字符串过长，还可以如下，以便易读(注意加括号)，系统自动拼接
msg = ('Put several strings within parentheses '
      'to have them joined together.')
```

```
msg = '请输入' + '字符' # 连接字符串
msg = "="*40           # 40 个等号分隔符=====
```

## 转义字符

7. 反斜杠'\'可以用来转义特殊字符，比如\n表示换行。
8. 使用字符前缀 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。

Escape Sequence	Meaning
<code>\newline</code>	续行符
<code>\\</code>	反斜杠符号 Backslash (\)
<code>\'</code>	单引号 Single quote (')
<code>\"</code>	双引号 Double quote (")
<code>\a</code>	响铃 ASCII Bell (BEL)
<code>\b</code>	退格 ASCII Backspace (BS)
<code>\f</code>	换页 ASCII Formfeed (FF)
<code>\n</code>	换行 ASCII Linefeed (LF)
<code>\r</code>	回车 ASCII Carriage Return (CR)
<code>\t</code>	横向制表符 ASCII Horizontal Tab (TAB)
<code>\v</code>	纵向制表符 ASCII Vertical Tab (VT)
<code>\oyy</code>	o 表示八进制
<code>\xhh</code>	x 表示十六进制

## 字符前缀

字符串前面还可以添加前缀，用来描述字符串应该如何解释。

```
stringprefix    ::= "r" | "u" | "R" | "U" | "f" | "F"
                  | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"

bytesliteral    ::= bytesprefix(shortbytes | longbytes)
bytesprefix     ::= "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb"
                  | "RB"
```



比如，如果字符串中带有转义字符，此时可以使用 `r` 来定义字符串，表示其后的字符串是 raw string，不需要转义。

```
>>> msg = "这是一个带有\n的字符串"
>>> print(msg)
这是一个带有
的字符串
>>> msg = r"这是一个带有\n的字符串"
>>> print(msg)
这是一个带有\n的字符串

#f 或 F 前缀表示格式化字符串
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
#如果不带前缀，则只会打印原串'Results of the {year} {event}'
```

## 字符串访问

9. 字符串为不可变数据类型，不能够修改其中的部分字符。
10. 字符串为序列变量，可以遍历访问。
11. 也可以使用下标索引访问，下标有两种方式：
  - a) 从左往右以 0 开始
  - b) 从右往左以 -1 开始
12. 截取子字符串的语法格式：变量[头下标:尾下标:步长]

```
str='Runoob'

print(str)                # 输出字符串 Runoob
print(str[0:-1])          # 输出第一个到倒数第二个的所有字符 Runoo
print(str[0])             # 输出字符串第一个字符 R
print(str[2:5])           # 输出从第三个开始到第五个的字符 noo
print(str[2:])            # 输出从第三个开始的后的所有字符 noob
str[0] = 'R'              # 这样会报错
```

## 字符串方法

<https://docs.python.org/3/library/string.html>

序号	方法及描述
1	<code>capitalize()</code> 将字符串的第一个字符转换为大写
2	<code>center(width, fillchar= ' ')</code> 返回一个指定宽度 <code>width</code> 居中的字符串, <code>fillchar</code> 为填充的字符, 默认为空格。
3	<code>count(str, beg= 0,end=len(string))</code> 返回 <code>str</code> 在 <code>string</code> 里面出现的次数, 如果 <code>beg</code> 或者 <code>end</code> 指定则返回指定范围内 <code>str</code> 出现的次数
4	<code>bytes.decode(encoding="utf-8", errors="strict")</code> Python3 中没有 <code>decode</code> 方法, 但我们可以使用 <code>bytes</code> 对象的 <code>decode()</code> 方法来解码给定的 <code>bytes</code> 对象, 这个 <code>bytes</code> 对象可以由 <code>str.encode()</code> 来编码返回。
5	<code>encode(encoding='UTF-8',errors='strict')</code> 以 <code>encoding</code> 指定的编码格式编码字符串, 如果出错默认报一个 <code>ValueError</code> 的异常, 除非 <code>errors</code> 指定的是 <code>'ignore'</code> 或者 <code>'replace'</code>
6	<code>endswith(suffix, beg=0, end=len(string))</code> 检查字符串是否以 <code>obj</code> 结束, 如果 <code>beg</code> 或者 <code>end</code> 指定则检查指定的范围内是否以 <code>obj</code> 结束, 如果是, 返回 <code>True</code> , 否则返回 <code>False</code> .
7	<code>expandtabs(tabsize=8)</code> 把字符串 <code>string</code> 中的 <code>tab</code> 符号转为空格, 默认的空格数是 8 。

8	<code>find(str, beg=0 end=len(string))</code> 检测 <code>str</code> 是否包含在字符串中，如果指定范围 <code>beg</code> 和 <code>end</code> ，则检查是否包含在指定范围内，如果包含返回开始的索引值，否则返回-1
9	<code>index(str, beg=0, end=len(string))</code> 跟 <code>find()</code> 方法一样，只不过如果 <code>str</code> 不在字符串中会报一个异常.
10	<code>isalnum()</code> 如果字符串至少有一个字符并且所有字符都是字母或数字则返回 <code>True</code> , 否则返回 <code>False</code>
11	<code>isalpha()</code> 如果字符串至少有一个字符并且所有字符都是字母则返回 <code>True</code> , 否则返回 <code>False</code>
12	<code>isdigit()</code> 如果字符串只包含数字则返回 <code>True</code> 否则返回 <code>False</code> ..
13	<code>islower()</code> 如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 <code>True</code> ，否则返回 <code>False</code>
14	<code>isnumeric()</code> 如果字符串中只包含数字字符，则返回 <code>True</code> ，否则返回 <code>False</code>
15	<code>isspace()</code> 如果字符串中只包含空白，则返回 <code>True</code> ，否则返回 <code>False</code> .
16	<code>istitle()</code> 如果字符串是标题化的(见 <code>title()</code> )则返回 <code>True</code> ，否则返回 <code>False</code>
17	<code>isupper()</code>

	如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
18	<code>join(seq)</code> 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	<code>len(string)</code> 返回字符串长度
20	<code>ljust(width[, fillchar])</code> 返回一个原字符串左对齐,并使用 fillchar 填充至长度 width 的新字符串，fillchar 默认为空格。
21	<code>lower()</code> 转换字符串中所有大写字符为小写.
22	<code>lstrip()</code> 截掉字符串左边的空格或指定字符。
23	<code>maketrans()</code> 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	<code>max(str)</code> 返回字符串 str 中最大的字母。
25	<code>min(str)</code> 返回字符串 str 中最小的字母。
26	<code>replace(old, new [, max])</code>

	把 将字符串中的 <code>str1</code> 替换成 <code>str2</code> , 如果 <code>max</code> 指定, 则替换不超过 <code>max</code> 次。
27	<code>rfind(str, beg=0, end=len(string))</code> 类似于 <code>find()</code> 函数, 不过是从右边开始查找.
28	<code>rindex( str, beg=0, end=len(string))</code> 类似于 <code>index()</code> , 不过是从右边开始.
29	<code>rjust(width, [, fillchar])</code> 返回一个原字符串右对齐, 并使用 <code>fillchar</code> (默认空格) 填充至长度 <code>width</code> 的新字符串
30	<code>rstrip()</code> 删除字符串字符串末尾的空格.
31	<code>split(str="", num=string.count(str))</code> <code>num=string.count(str))</code> 以 <code>str</code> 为分隔符截取字符串, 如果 <code>num</code> 有指定值, 则仅截取 <code>num+1</code> 个子字符串
32	<code>splitlines([keepends])</code> 按照行 (' <code>\r</code> ', ' <code>\r\n</code> ', ' <code>\n</code> ') 分隔, 返回一个包含各行作为元素的列表, 如果参数 <code>keepends</code> 为 <code>False</code> , 不包含换行符, 如果为 <code>True</code> , 则保留换行符。
33	<code>startswith(substr, beg=0, end=len(string))</code> 检查字符串是否是以指定子字符串 <code>substr</code> 开头, 是则返回 <code>True</code> , 否则返回 <code>False</code> 。如果 <code>beg</code> 和 <code>end</code> 指定值, 则在指定范围内检查。
34	<code>strip([chars])</code> 在字符串上执行 <code>lstrip()</code> 和 <code>rstrip()</code>
35	<code>swapcase()</code>

	将字符串中大写转换为小写，小写转换为大写
36	<code>title()</code> 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 <code>istitle()</code> )
37	<code>translate(table, deletechars="")</code> 根据 <code>str</code> 给出的表(包含 256 个字符)转换 <code>string</code> 的字符，要过滤掉的字符放到 <code>deletechars</code> 参数中
38	<code>upper()</code> 转换字符串中的小写字母为大写
39	<code>zfill (width)</code> 返回长度为 <code>width</code> 的字符串，原字符串右对齐，前面填充 0
40	<code>isdecimal()</code> 检查字符串是否只包含十进制字符，如果是返回 <code>true</code> ，否则返回 <code>false</code> 。

### 3. 列表 List

<https://docs.python.org/3/library/stdtypes.html#lists>

#### 列表定义

列表是用 `[]` 来表示，用逗号分隔开元素列表

列表中的元素的类型可以不相同，也可嵌套高级数据类型。

```
lt = [ 'ab' , 12, 2.34, (5,6,7)]      #1. 直接定义列表
print(lt)
print( '列表长度=' , len(lt))
```

```

lt = [] #2. 空列表
LNUM = 5
lt = [0]*5 #3. 元素全为 0 的列表[0, 0, 0, 0, 0, 0]
lt = [I for I in range(LNUM)] #4. 范围列表[1, 2, 3, 4, 5]

lt = list() #5. 空列表
lt = list(range(1, 5)) #6. 范围列表
lt = list('abc') #7. 将字符中转换为列表 [ 'a' , ' b' , ' c' ]
lt = list((1, 2, 3)) #8. 将元组转换为列表[1, 2, 3]
lt = [['a', 'b', 'c'], [1, 2, 3], 4] #9. 列表嵌套

```

## 列表访问

列表访问有两种方式：

1. 采用索引下标访问。
  - a) 从左往右以 0 开始
  - b) 从右往左以 -1 开始
2. 遍历元素（for）。

```

lt = ['a', 'b', 'c']
print(lt[0]) #打印第一个元素
print(lt[-1]) #打印最后一个元素

#遍历元素
for itm in lt:
    print(itm)

#遍历索引和元素值
for index, val in enumerate(lt):
    print('索引=' , index, ' 值=' , val)

#按值来寻找索引下标
index = lt.index('b')

```

## 列表修改

```
lt[0] = 'e'          #将第一个元素修改为'e'
print(lt)

lt.append('f')        #在最后添加一个元素
lt.extend(['g', 'h']) #添加多个元素, 相当于添加列表
lt.insert(0, 'OK')    #指定位置插入新元素

itm = lt.pop()        #删除末尾的元素, 返回的元素可以继续使用
itm = lt.pop(-1)      #删除指定位置的元素, 并返回该元素

itm = lt.remove('a')  #删除指定值的元素, 即按值删除
                    #如果值不在列表中, 会抛 ValueError 异常
                    #如果存在相同值的多个元素, 则只删除第一个

del(lt[0])            #删除指定位置的元素
del(lt[0:2])          #删除多个元素, 列表切片

lt.clear()            #清空整个列表
```

## 列表操作

### 列表的排序、反转

```
#列表合并
lt1 = list('abc')
lt2 = list('efg')
lt = lt1 + lt2        #两个列表合并
lt1.extend(lt2)       #将 lt2 合并到 lt1 列表中

cars = ['BMW', 'Audi', 'Toyota', 'Subaru']
cars.sort()           #列表升序
cars.sort(reverse=True) #列表降序
cars.reverse()        #列表反转顺序

tmp = sorted(cars)     #临时排序, 不改变原来列表
print('原始列表:', cars)
print('排序后列表:', tmp)

newCars = cars.copy()  #列表复制
newCars = cars[:]      #列表复制, 参考列表切片
```



```

lt = list( 'abc' )
newlt = lt                                #列表指针赋值，指向同一个列表，不是复制
lt[0]=' d'
print(newlt)                             #改变 lt,newlt 也改变了[ 'd' , ' b' , ' c' ]

#列表其它常用函数
a = list(range(1,5))
print(len(a),max(a),min(a))

num = cars.count( 'Audi' )                #查看值在列表中出现的次数
indx = cars.index( 'Audi' )               #查看元素第一次出现的位置

```

## 列表切片

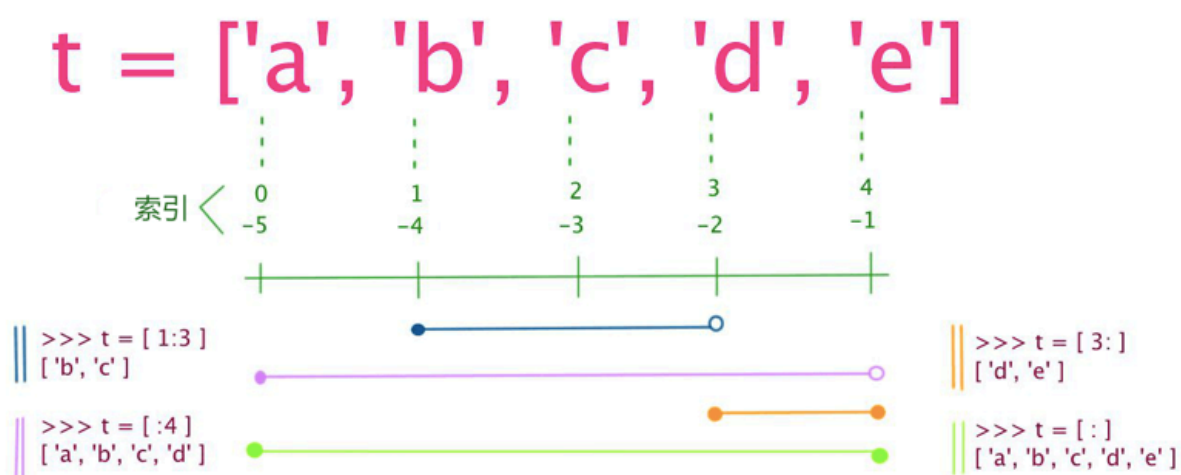
列表切片，只选择列表的部分元素。

列表和字符串一样，列表切片形如：变量[i:j:k]，表示选择元素的从头下标 i 开始，间隔为 k，到尾下标 j，但不包括尾下标 j 的元素。

即子集元素索引 x 满足： $x=i+n*k$ ， $n \geq 0$  且  $i \leq x < j$ 。

列表切片不修改原列表，只返回被截取后的新列表。

列表截取的语法格式如下：




```
lt = list( 'abcde' )
newlt = lt[:]          #复制整个列表,
newlt = lt[0:2] #返回[ 'a' , ' b' ]
newlt = lt[2:]  #返回[ ' c' , ' d' , ' e' ], 默认尾下标到结尾
newlt = lt[:2]  #返回[ 'a' , ' b' , ' c' ], 默认起始下标从 0 开始
newlt = lt[4:2] #返回空列表, 当 j<=i 时都是空列表

lt = list(range(10))
newlt = lt[1:9:2]      #返回[1, 3, 5, 7]
```

0
1
2
3
4
5
6

```
>>> letters = [ 'c', 'h', 'e', 'c', 'k', 'i', 'o' ]
```



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

## 列表方法

<https://docs.python.org/3/library/stdtypes.html#lists>

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	<code>list.insert(index, obj)</code> 将对象插入列表
6	<code>list.pop([index=-1])</code> 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	<code>list.remove(obj)</code> 移除列表中某个值的第一个匹配项
8	<code>list.reverse()</code> 反向列表中元素
9	<code>list.sort(cmp=None, key=None, reverse=False)</code> 对原列表进行排序
10	<code>list.clear()</code> 清空列表
11	<code>list.copy()</code> 复制列表

## 4. 元组 Tuple

元组是不可改变的列表，除了不可修改元素外，其余操作与列表基本一样。

### 元组定义

1. 元组是用 `()` 来表示，用逗号分隔开元素列表。
2. 元组中的元素类型可以不相同，也可嵌套高级数据类型。
3. 元组与字符串类似，其实，可以把字符串看作一种特殊的元组。
4. 元组和字符串一样，可以使用 `+` 进行连接

```

tp = 12, 34, 'hello'
tp = ('ab', 12, [1, 2, 3])
tp = ()

tp = (20,)

```

#1. 直接定义  
#2. 空元组  
#3. 一个元素的元组，注意：必须添加逗号

```

print(type(tp))      #<class 'tuple'>

tp = (20)             #这样定义的话，返回的就是 int 型
print(type(tp))      #<class 'int'>

tp = tuple('abcd')    #4. 字符串转换为元组
tp = tuple(i for i in range(10)) #5. 序列转换为元组
tp = tuple([1, 2, 'OK']) #6. 列表转换为元组

tp = ('Hi',)*4        #7. 有 4 个元素的元组，所有元素初始化为
                      #'Hi'，即重复 4 次的元素
                      #('Hi', 'Hi', 'Hi', 'Hi')

tp1 = ('ab', 12)
tp2 = (2, 'cd')
tp = tp1+tp2          #元组合并为('ab, 12, 2, ' cd' )
print(tp)

```

## 元组访问

- 元组的访问和列表一样，可以使用索引。从左往右以 0 开始，从右往左以 -1 开始。
- 元组也可以使用遍历语句。
- 元组也可以使用切片变量[i:j:k]

```

tp = ('ab', 12, 'cd') #索引访问元素
print(tp[0])          #' ab'
print(tp[-1])         #' cd'

for itm in tp:         #遍历元素
    print(itm)

#遍历索引和元素值
for index, val in enumerate(tp):
    print('索引=' , index, ' 值=' , val)

print(tp[0:2])        #参考列表的切片

```

## 元组修改

- 元组中的元素类型不可以修改，但可以给整个元组变量重新赋值。
- 但元组可以包含可变的对象，比如 list 列表，此时可以修改元素列表中的值。

```
>>>t = [1, 2, 3]
>>>tp = ('ab', 12, t)
>>>tp[0] = 'cd'                                     #修改元素，报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>>tp = ('cd', 34, t)                                #变量重新赋值
>>>tp[2][0] = 4                                       #修改元素列表中的值
>>>print(tp)
('cd', 34, [4, 2, 3])
```

## 5. 字典 dictionary

<https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

字典和列表类似，两者区别在于：

- 列表是有序的对象集合，而字典是无序的对象集合。
- 列表是通过索引存取，而字典中元素通过键来存取。
- 列表的索引必须是数字，但字典中的键却可以是字符串。
- 列表的索引是隐式的，而字典保留的是 key-Value 对

## 字典定义

- 字典用 {} 标识，用逗号分隔元素，用 : 分隔键值（键-值）。
- 键(key)必须是不可变类型，如字符串，数字或元组等。

3. 键必须是唯一的，不能重复；重复的关键字指的是同一个元素。

```
dct = {} #1. 空字典
dct = { 'color' : 'green' , 'point' : 5} #2. 键值映射对构造

#3. 形参格式定义
dct = dict(color='green' , point=5)

#4. 单列表构造，列表中元素为键值元组
dct = dict([('color','green'), ('point',5)])

#5. 两个列表构造，第一个列表为键列表，第二个列表为值列表
dct = dict.fromkeys(['color' , 'point' ], ['green' , 5])

#6. 两个列表构造，第一个列表为键列表，第二个列表为值列表
#zip 函数返回的是键值元组形成的列表，与 4 的定义是相同的
dict = dict(zip(['color','point'], ['green',5]))

#7. 键值映射对构造，返回 {2: 4, 4: 16, 6: 36}
dct2 = {x: x**2 for x in (2,4,6)}
```

## 字典访问

字典访问有两种方式：

4. 采用键值访问。
5. 遍历元素（for）。

```
>>> dct = {'color': 'green', 'point': 5}
>>> dct['color'] #如果 key 不存在，则抛出异常
'green'
>>> cl = dct.get('color') #如果 key 不存在，默认返回 None，不抛出异常
'green'
>>> dct['point']
5
>>> dct
{'color': 'green', 'point': 5}
```

```

#单独遍历键列表，dct 默认返回键列表
for itm in dct:
    print(itm)

for itm in dct.keys():
    print(itm)

#单独遍历值列表
for itm in dct.values():
    print(itm)

#同时遍历键和值列表
for key, val in dct.items():
    print( '键=' ,key, ' 值=' , val)

```

## 字典修改

字典是动态的数据类型，可以修改

```

dct = {'color':'green', 'point':5}
dct[ 'loc' ] = (23,45)           #添加元素(键值对)
dct[ 'color' ] = 'red'           #修改元素
dct.update(red=1,blue=2)         #修改/增加元素

#按键删除元素
del dct[ 'loc' ]                 #键不存在，则抛出 KeyErro 异常
del(dct[ 'loc' ])                #同上一样

#按键弹出元素
cl = dct.pop( 'color' )          #删除指定键值元素，返回元素值

#自动删除末尾元素
cl = dct.popitem()               #删除末尾的键值对

dct.clear()                      #清空整个字典

```

## 字典操作

```
#判断键是否在字典中
if key in dct
if key in dct.keys()

#判断值是否在字典中
if val in dct.values()

#判断键值对是否在字典中
if (key,val) in dct.itmes()

l = len(dct)                #字典长度

#字典复制
dct1 = {'one':1, 'two':38, 'three':'red'}
dct2 = dct1.copy()

dct = dct1                  #指针赋值，两个字典指向的是同一个字典
                             #修改其中一个，另一个同样变化。
```

## 字典函数和方法

Python 中的列表有如下内置函数处理：

len(dict) 返回字典元素个数

str(dict) 返回字典元素

字典包含以下方法：

序号	函数及描述
1	<a href="#"><u>radiansdict.clear()</u></a> 删除字典内所有元素
2	<a href="#"><u>radiansdict.copy()</u></a> 返回一个字典的浅复制



3	<u><code>radiansdict.fromkeys()</code></u> 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	<u><code>radiansdict.get(key, default=None)</code></u> 返回指定键的值，如果值不在字典中返回 default 值
5	<u><code>key in dict</code></u> 如果键在字典 dict 里返回 true，否则返回 false
6	<u><code>radiansdict.items()</code></u> 以列表返回可遍历的(键，值) 元组数组
7	<u><code>radiansdict.keys()</code></u> 返回一个迭代器，可以使用 list() 来转换为列表
8	<u><code>radiansdict.setdefault(key, default=None)</code></u> 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	<u><code>radiansdict.update(dict2)</code></u> 把字典 dict2 的键/值对更新到 dict 里
10	<u><code>radiansdict.values()</code></u> 返回一个迭代器，可以使用 list() 来转换为列表
11	<u><code>pop(key[, default])</code></u> 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	<u><code>popitem()</code></u> 随机返回并删除字典中的一对键和值(一般删除末尾对)。

## 6. 集合 set

<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>

集合是一个无序的不重复的元素序列。

### 集合定义

1. 使用 {} 定义集合，元素间用逗号分隔。
2. 集合中的元素不允许重复，重复的元素当成一个，且没有顺序。

```
>>>st = set()           #1. 空集合，注意不能用 {} 来定义空集合. {} 表示空字典
>>>st = {1, 2, 2, 4}    #2. 序列定义集合
{1, 2, 4}

>>>st = set('abcd')     #3. 字符串定义集合
>>>st = {x for x in 'abcd'}           #4. 序列
>>>st = {x for x in 'abcd' if x not in 'abef'}
{'c', 'd'}

st = set([1, 2, 3, 4])   #5. 列表转化为集合
st = set((1, 2, 'OK'))  #6. 元组转化为集合
```

### 集合访问

集合不支持索引，只能遍历

```
#集合不支持索引，只能遍历
st = set([1, 3, 5])
for val in st:
    print(val)
```

## 集合修改

```
#添加元素，元素必须是可哈希的
st.add(6)          #添加元素，元素为数字
st.add('OK')       #添加元素，元素为字符串
st.add((1, 2, 3))  #添加元素，此元素为元组
#st.add({2, 'Hello'}) #TypeError 异常，不能添加字典元素
#st.add([4, 5, 6])  #TypeError 异常，不能添加列表元素

#添加多个元素，从其它迭代类中添加新的集合元素
#注意：add 函数与 update 函数的区别
st.update('abc')    #将单个字符增加到集合中
st.update([4, 'OK']) #将列表中元素增加到集合中
                    #即将列表中的单个元素加入集合，并不是把整个列表当成一个元素加入集合
st.update((1, 6))   #将元组中元素增加到集合中
st.update({'a':5, 'b':9}) #将字典中的所有键增加到集合中，注意：字典中的值没有用

#添加混合元素
st = set([1, 3, 5])
st.update((4, 5), {'a':5, 'b':9}, 'bc') # {1, 3, 4, 5, 'c', 'b', 'a'}

#集合删除
st.remove(3)        #删除指定元素，移除不存在的值时，会抛出 KeyError 异常
st.discard(3)       #丢弃指定元素，移除不存在的值时不会报错

val = st.pop()      #弹出某个元素，并返回此元素。集合为空时会抛出 KeyError 异常

st.clear()          #清空集合，st 为空集合
```

## 集合运算

在数学中，集合有并集、交集、补集/差集、（反集？）的运算。

```
>>>a = set('abcd')
>>>b = set('cdef')
```

```

#集合运算
>>>c = a|b          #并集 A∪B
{ 'a' , 'b' , 'c' , 'd' , 'e' , 'f' }

>>>c = a&b          #交集 A∩B
{ 'c' , 'd' }

>>>c = a - b         #差集/补集 B - A = { x | x∈B 且 x∉A}
{ 'a' , 'b' }

>>>c = a ^ b         #不包含 a 和 b 的公共元素
{ 'a' , 'b' , 'e' , 'f' }

```

## 集合函数和方法

Python 中的列表有如下内置函数处理：

len(set) 返回集合元素个数

max(tuple) 返回最大元素

min(tuple) 返回最小元素

set(seq) 将序列转换为集合

方法	描述
<u><a href="#">add()</a></u>	为集合添加元素
<u><a href="#">clear()</a></u>	移除集合中的所有元素
<u><a href="#">copy()</a></u>	拷贝一个集合
<u><a href="#">difference()</a></u>	返回多个集合的差集
<u><a href="#">difference_update()</a></u>	移除集合中的元素，该元素在指定的集合也存在。

<u><code>discard()</code></u>	删除集合中指定的元素
<u><code>intersection()</code></u>	返回集合的交集
<u><code>intersection_update()</code></u>	删除集合中的元素，该元素在指定的集合中不存在。
<u><code>isdisjoint()</code></u>	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。
<u><code>issubset()</code></u>	判断指定集合是否该方法参数集合的子集。
<u><code>issuperset()</code></u>	判断该方法的参数集合是否为指定集合的子集
<u><code>pop()</code></u>	随机移除元素
<u><code>remove()</code></u>	移除指定元素
<u><code>symmetric_difference()</code></u>	返回两个集合中不重复的元素集合。
<u><code>symmetric_difference_update()</code></u>	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。
<u><code>union()</code></u>	返回两个集合的并集
<u><code>update()</code></u>	给集合添加元素

## 类型转换

下面的函数可以实现数据类型的转换。

函数	函数描述
1. <code>int(x [, base])</code>	将 x 转换为一个整数
2. <code>float(x)</code>	将 x 转换到一个浮点数
3. <code>complex(real [, imag])</code>	创建一个复数
4. <code>str(x)</code>	将对象 x 转换为字符串
5. <code>repr(x)</code>	将对象 x 转换为表达式字符串
6. <code>eval(str)</code>	用来执行表达式，并返回表达式的值
7. <code>tuple(s)</code>	将序列 s 转换为一个元组
8. <code>list(s)</code>	将序列 s 转换为一个列表
9. <code>set(s)</code>	转换为可变集合
10. <code>dict(d)</code>	创建一个字典。d 必须是一个序列 (key, value) 元组。
11. <code>frozenset(s)</code>	转换为不可变集合
12. <code>chr(x)</code>	将一个整数转换为一个字符
13. <code>ord(x)</code>	将一个字符转换为它的整数值

14. <code>hex(x)</code>	将一个整数转换为一个十六进制字符串
15. <code>oct(x)</code>	将一个整数转换为一个八进制字符串

## 7. 日期时间

时间相关的类型，是通过模块 `datetime` 提供的。常用的时间类有四个：`datetime`, `date`, `time`, `timedelta`。

### 日期时间 *datetime*

这个是最全面的时间类，用来表示（年、月、日、时、分、秒、微秒）。

此类还提供了获取当前时间、将时间格式化等其它函数。

### 日期 *date*

`date` 对应日期（年、月、日），只用来表示 `datetime` 中的前一部分 `date` 部分

### 时间 *time*

`time` 对应一天当中的时间（时、分、秒、微秒），只用来表示 `datetime` 中的后一部分 `time` 部分。

## 时间间隔 *timedelta*

`timedelta` 是一个表示时间段的对象，表示两个 `datetime` 时间之间的差。

## 时区 *timezone*

时区，表示的是本地时区与 UTC 的偏移时间，取值范围（-24~24 小时）

## 特殊类型使用

其它的特殊类型，比如 `range` 类，以及迭代类、生成器、序列类等。

## *range* 类

很奇怪，在 Python 专门有一个 `range` 类。它定义了一个整数序列，可以使用 `for` 语句来遍历。`range` 类的定义格式如下：

```
range([start, ]stop[, step])
```

其中参数均要求为整数，`stop` 是必选参数，其余是可选的，`start` 默认为 0，`step` 默认为 1。其整数元素 `x` 满足如下要求：

- 1)  $\text{start} \leq x < \text{stop}$
- 2)  $x = \text{start} + n * \text{step}$ , `n` 为非负整数。

显然，当 `start >= stop` 时，`range` 类为空。

```
>>> rg = range(5)
>>> rg
range(0, 5)
>>> type(rg)
<class 'range'>
>>> dir(range)
['__bool__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
```



```
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'count', 'index', 'start', 'step', 'stop']
>>> for num in rg:
    print(num)
```

## 迭代器 *iterator*

迭代器用于遍历数据集的元素，它可以记住上次访问的数据的位置。迭代器对象从数据的第一个元素开始访问，直到所有的元素访问结束。

支持迭代器功能的类，也叫做迭代类。

类似字符串、列表、元组等都可用于创建迭代器。

1. 迭代对象可以使用 `iter()` 和 `next()` 来访问元素。
2. 也可以使用 `for` 语句遍历元素。
3. 创建自己的迭代类时，必须实现 `__iter__()` 和 `__next__()` 两个特殊的方法。

**#很多内置数据类型都支持迭代器**

```
>>> lt = list('abc')
>>> type(lt.__iter__())
<class 'list_iterator'>
```

```
>>> s = 'OK'
>>> type(s.__iter__())
<class 'str_iterator'>
```

```
>>> dt = {'a':1, 'b':'end'}
>>> type(dt.__iter__())
<class 'dict_keyiterator'>
```

**#=====迭代类可以使用 `iter()` 和 `next()` 来访问=====**

**#=====当没有元素时，抛出 `StopIteration` 异常**

```
>>> lt = [1,2]
>>> it = iter(lt)           #创建迭代器
>>> next(it)               #1. 使用 next 访问下一个元素
```

```

1
>>> next(it)
2
>>> next(it)          #遍历到最后，抛出 StopIteration 异常，需要自己处理
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

#=====迭代类也可以使用 for 循环来遍历=====
>>> s = 'OK'
>>> for char in s:
...     print(char)

>>> lt = [1, 2, 3]
>>> lt = [1, 2, 3]
>>> for num in lt:      # For 循环会自动处理 StopIteration 并结束
...     print(num)

```

```

#创建自己的迭代类，假定是元素是元组
class MyData:
    def __init__(self):
        self.tuplist = [(1, 2), (3, 4), (5, 6)]      #假定是固定的列表
        self.loc = 0
    def __iter__(self):          #返回迭代器本身即可
        return self
    def __next__(self):
        if self.loc == len(self.tuplist):
            self.loc = 0          #这句很重要哟
            raise StopIteration
        else:
            tup = self.tuplist[self.loc]
            self.loc += 1
            return tup

#1. 使用迭代器类
myclass = MyData()
it = iter(myclass)
for x in it:
    print(x)

#2. 或者直接使用类, 也是可以的
for x in myclass:
    print(x)

```

```
#3. 也可以使用 next 读取
it = iter(mydata)
while True:
    try:
        itm = next(it)
    except StopIteration as e:
        break
    else:
        print(itm)
```

## 生成器 *yield*

生成器是一个返回迭代器的函数，使用关键字 `yield` 返回迭代元素。

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。

### #生成器函数

```
def Fibonacci(n):                #定义一个生成器函数—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
```

### #使用生成器函数

#### #1) 直接遍历

```
it = Fibonacci(10)
for x in it:
    print(x, end = ' ')
print('==end==')
```

#### #2) 使用 next 函数遍历

```
it = Fibonacci(10)            #注意：这里要重新赋值，因为前面代码 it 已经到末尾了
while True:
    try:
        print(next(it), end=' ')
```

```

except StopIteration:
    break
print('==end==')

```

## 序列类 *Sequence*

系统定义了 7 种序列类（其实并没有一个专门的类型叫 sequence）。序列类分为两大类：

- 1) 不可变序列类：str, range, tuple, bytes
- 2) 可变序列类：list, bytearray, memoryview

序列类有一些共同的特征，所有序列类支持如下操作：

Operation	Result
<code>x in s</code>	True if an item of <i>s</i> is equal to <i>x</i> , else False
<code>x not in s</code>	False if an item of <i>s</i> is equal to <i>x</i> , else True
<code>s + t</code>	the concatenation of <i>s</i> and <i>t</i>
<code>s * n</code> Or <code>n * s</code>	equivalent to adding <i>s</i> to itself <i>n</i> times
<code>s[i]</code>	<i>i</i> th item of <i>s</i> , origin 0
<code>s[i:j]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i>
<code>s[i:j:k]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> with step <i>k</i>
<code>len(s)</code>	length of <i>s</i>
<code>min(s)</code>	smallest item of <i>s</i>
<code>max(s)</code>	largest item of <i>s</i>
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <i>x</i> in <i>s</i> (at or after index <i>i</i> and before index <i>j</i> )
<code>s.count(x)</code>	total number of occurrences of <i>x</i> in <i>s</i>

```

>>> "gg" in "eggs"
True

```

可改变的序列支持如下操作：

Operation	Result
<code>s[i] = x</code>	item <i>i</i> of <i>s</i> is replaced by <i>x</i>
<code>s[i:j] = t</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> is replaced by the contents of the iterable <i>t</i>
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <i>t</i>
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list
<code>s.append(x)</code>	appends <i>x</i> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code> )
<code>s.clear()</code>	removes all items from <i>s</i> (same as <code>del s[:]</code> )
<code>s.copy()</code>	creates a shallow copy of <i>s</i> (same as <code>s[:]</code> )
<code>s.extend(t)</code> Or <code>s += t</code>	extends <i>s</i> with the contents of <i>t</i> (for the most part the same as <code>s[len(s):len(s)] = t</code> )
<code>s *= n</code>	updates <i>s</i> with its contents repeated <i>n</i> times
<code>s.insert(i, x)</code>	inserts <i>x</i> into <i>s</i> at the index given by <i>i</i> (same as <code>s[i:i] = [x]</code> )
<code>s.pop([i])</code>	retrieves the item at <i>i</i> and also removes it from <i>s</i>
<code>s.remove(x)</code>	remove the first item from <i>s</i> where <code>s[i]</code> is equal to <i>x</i>
<code>s.reverse()</code>	reverses the items of <i>s</i> in place

## 上下文管理器 *with*

对于一些特殊的系统资源，比如文件、数据连接、socket 等，这些资源在应用程序打开并使用后，必须要做的一件事就是关闭（断开）该资源（即相当于释放资源占用锁），以便其他程序使用。但是，在大多数情况下，如果程序发生异常，很有可能会跳过资源释放的代码，导致资源被锁死。所以，常规地，往往需要在代码中加上较多的异常处理语句，导致代码复杂程度较高。

而 Python 中的上下文管理器（ContextManager），就可以比较优雅地解决这个问题，与此相关的语句主是 `with-as` 语句。

```
with expression [as variable]:  
    with-block
```

- 1) 上下文管理器就是实现了上下文协议的类，而上下文协议就是一个类要实现 `__enter__()` 和 `__exit__()` 两个系统方法。
- 2) `__enter__()` 方法会在执行 `with` 后面的语句时执行，一般用来处理操作前的内容，比如文件打开，数据连接等。

`__enter__()` 需要返回待使用的资源对象。

- 3) `__exit__()` 方法会在 `with` 内的代码执行完毕后执行，一般用来处理一些善后收尾工作，比如文件的关闭，数据库断开等。

完整形式为 `__exit__(type, value, traceback)`，这三个参数和调用 `sys.exec_info()` 函数返回值是一样的，分别为异常类型、异常信息和堆栈。如果执行体语句没有引发异常，则这三个参数均被设为 `None`。否则，它们将包含上下文的异常信息。

`__exit__()` 方法返回 `True` 或 `False`，分别指示被引发的异常有没有被处理，如果返回 `False`，引发的异常将会被传递出上下文。如果 `__exit__()` 函数内部引发了异常，则会覆盖掉执行体的中引发的异常。处理异常时，不需要重新抛出异常，只需要返回 `False`，`with` 语句会检测 `__exit__()` 返回 `False` 来处理异常。

- 4) `with` 语句不仅可以管理文件，还可以管理锁、连接等等。

## with 语句

```
with open("text.txt") as f:  
    for line in f.readlines():  
        print(line)  
#此时，程序会自动调用 f.close() 语句
```

```
#管理锁
import threading
lock = threading.Lock()
with lock:
    #执行一些操作
    pass
```

## 自定义管理类

5) 可以自定义一个上下文管理器。

```
class DBConnection(object):
    def __init__(self):
        pass

    def cursor(self):
        #返回一个游标并且启动一个事务
        pass

    def commit(self):
        #提交当前事务
        pass

    def rollback(self):
        #回滚当前事务
        pass

    def __enter__(self):
        #返回一个 cursor
        cursor = self.cursor()
        return cursor

    def __exit__(self, type, value, tb):
        if tb is None:
            #没有异常则提交事务
            self.commit()
        else:
            #有异常则回滚数据库
```

```

        self.rollback()

sql = 'select * from User'
con = DBConnection()
with con as cursor:
    cursor.execute(sql)
    #...

```

## contextManage 对象

前面提到，实现上下文管理器需要实现自定义实现两个系统方法的类，还是比较麻烦的，如果采用装饰器来实现会更简单。Contextlib 模块包含一个装饰器和一些辅助函数，只需要写一个生成器函数就可以代替上下文管理器。

- 1) 生成器函数在 yield 之前的代码等同于 \_\_enter\_\_() 函数；
- 2) yield 的返回值等同于 \_\_enter\_\_() 函数的返回值。

```

from contextlib import contextmanager

@contextmanager
def my_open(path, mode):
    f = open(path, mode)
    yield f
    f.close()

with my_open( 'mytext.txt' , 'r' ) as f:
    #文件操作

#此处自动调用 f.close() 函数

@contextmanager
def transaction(db):
    db.begin()
    try:
        yield
    except:
        db.rollback()
        raise

```



```
else:  
    db.commit()
```

## 五、函数类模块包

### 自定义函数

函数是可重复使用的用于实现单一功能的代码段。函数提高了代码的可重用性，也方便了组织代码结构。

Python 中提供了内置函数（比如 open, max 等），也支持用户自定义函数。

### 函数定义

1. 自定义函数的格式如下：

```
def 函数名(参数列表):  
    函数体
```

2. 函数定义时可以有形参，也可以不带形参。

3. 形参可以是必须的，也可以是可选形参（即有默认值）。

4. 函数可以无返回（相当于默认返回 None），也可以有返回值，甚至可以返回多个值。

```
def hello1():  
    print("Hello World!")  
    #return None  
#1. 无参数，无返回。相当于 return None  
  
def hello2(msg):  
    print(msg)  
    return  
#2. 有参数，无返回
```

```
def hello3(msg=' Hello World' ):          #3. 有参数，参数可带默认值
    print(msg)
    return

#计算面积函数
def Area(width, height):                  #3. 有参数，有返回，返回单个值
    return width*height

def AreaandCircle(width, height):         #4. 有参数，有返回，返回多个值（面积和周长）
    area = width*height
    circle = 2*width*height
    return area, circle

def AreaandCircle(width, height, msg=" 计算完成" )    #5. 有默认值的形参位置要靠后
    area = width*height
    circle = 2*width*height
    print(msg)
    return area, circle
```

## 函数参数

5. 函数参数可以没有，也可以有，甚至可以有多个形参。
6. 函数的参数可以是必选参数，也可以有可选参数（即有默认值）。
7. 如果参数有默认值，其位置应该靠后。
8. 参数可以是任意数量的个数（\*varlist），加\*表示以元组的形式传入，存放所有未命名的变量参数。
9. 参数还可以是任意数量的关键字参数（即带参数名字 \*\*varlist），加两个星号\*\*，表示以字典的形式导入，在调用时需要带上形参名。

```
def fun1(*var):                          #1. 形参为可变参数，即元组
    print( " 输入的参数有： " )
```

```

    for itm in var:
        print(itm, end=' ')
    print()
    return

fun1(10, 30, ' 45' )

def fun2( **var):
    print( "输入的参数有: " )
    for key, val in var.items():
        print( 'key=', key, 'val=', val)
    return

fun2(name=' jane' , age=20, color=' msg' )

```

#打印一个换行

#输入多个参数

#2. 形参为可变参数，即字典

#以键值对输入字典参数

## 函数返回值

如果函数要向外（即调用者）传递处理后的数据，有两种做法：

10. 通过 return 返回数据，可以返回多个。
11. 通过可变的实参，实参可以在函数体内被修改。

#如果函数要向外传出数据，有两种做法：

#1. 使用 return 传回来。（如上 AreaandCircle 所示）

#2. 使用可变参数向外传（整数/字符串/元组是不可变的，而列表/字典/集合是可变的）

```

def fun( mylist):
    mylist.append([1, 2, 3])
    return

ls = [10, 20, 30]
print( "函数调用前列表: ", ls)
fun(ls)
print( "函数调用后列表: ", ls)

```

#形参为可变参数

#7. 通过可变列表传递

# 调用前和后的列表是同一个

## 函数调用

12. 函数调用时，默认情况下，传入的实参个数、位置必须与函数的形参相匹配。
13. 一般，按位置匹配的实参叫位置参数（Positional arguments），按键值对（kwarg = value）形式的实参叫关键字参数（keyword arguments）。这两种实参可以混用的。
14. 如果有形参有默认值，则可以不传入相应实参（实参个数少于形参个数）。
15. 函数调用时，如果采用关键字参数，则可以不考虑参数位置（位置无关）。

```
hello1()                                #1. 直接调用函数
hello2("OK")                            #2. 带参数调用函数
s = Area(2, 3)                          #3. 使用返回值
s, c = AreaandCircle(20, 10)           #4. 使用多个返回值
print('面积=', s)
print('周长=', c)

def printinfo(name, age):
    print("名字=", name)
    print("年龄=", age)

nm = '张三'; n = 25
printinfo(nm, n)                        #5. 默认按照顺序传递变量
printinfo(age=n, name=nm)              #6. 指定实参名称，可以不考虑顺序

#可以使用字典实参
>>> def parrot(voltage, state='a stiff', action='vroom'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.", end=' ')
    print("E's", state, "!")

>>> d = {"voltage": "four million", "state": "bleedin' demised", "action":
"VOOM"}
>>> parrot(**d)
```

## 匿名函数

正常情况下，函数是需要重复使用的，往往需要函数名来进行多次引用。然后，在某些情况下，比如函数只在一个地方使用一次，或者函数实现语句非常简单，此时独立定义一个函数是没有意义的。所以，Python 中支持匿名函数的定义和使用。

- 1) 在 python 中使用 lambda 来创建匿名函数。格式如下：  
Lambda 参数列表:表达式
- 2) 匿名函数中参数列表可选，只包含一个表达式语句，且不能访问全局参数。
- 3) 也可以给匿名函数取一个函数名。
- 4) 函数名可以当成参数传递。

```
#定义 lambda 匿名函数，并给它一个名字
sum = lambda arg1, arg2: arg1+arg2

total = sum(10, 20)           #调用函数

#匿名函数返回函数
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f(0)           #返回 42
f(1)           #返回 43

#匿名函数返回值
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[0])      #按第一个值排序
>>> pairs.sort(key=lambda pair: pair[1])      #按第二个值排序
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

## 变量作用域

变量的作用域决定了哪些代码块可以使用这个该变量名称。

### 四层变量作用域

Python 变量一共有 4 个层次的作用域：

## Python 四层作用域



- 1) L(local) 局部作用域，只在本代码块内有效
  - 典型的，函数体的变量都是局部变量
- 2) E(Enclosing) 闭包作用域。
  - 这是一种特殊的作用域，往往存在于嵌套函数的场景。

- 在外层函数内的一个局部变量，相对于内层函数来说，就是一个闭包变量。
  - （这种场景较少使用，函数内再嵌套一个函数，谁会发神经这样写呢？）
- 3) G(Global)全局作用域，模块内或导入该模块的程序都可以访问
- 典型的，就是在函数体外的该模块内的变量就是全局变量。
- 4) B(Built-in)内置作用域，任何程序皆可直接访问
- 即 Python 内置函数所在模块的范围，不需要明确导入。
  - 典型的系统属性如\_\_name\_\_，\_\_sepc\_\_等。

## 作用域使用规则

不同作用域的变量，在使用时应该遵循如下规则：

- 1) 本层作用域可直接读取或修改本层定义的所有变量。
- 2) 下层作用域可读取上层变量，但不能修改（赋值）上层变量。实际上，在下层作用域内赋值，相当于在下层重新定义了一个名称相同的新的变量，从而覆盖了上层变量的作用域。
- 3) 在下一层作用域内，可以重新定义一个与上层变量名相同的变量名，且互不影响。
- 4) 当在不同层次有相同变量名时，在使用变量时，变量的查找顺序是由本层向高层依次查找。完整的顺序是：L→E→G→B，即在局部找不到，就会去局部外的局部去找，再找不到就去全局找，最后在内置模块中去找。如果都找不到，才会抛出 NameError 异常。

```
count = 0                #G 全局作用域，当前整个模块内可使用
def outer():
    print(count)
    ecount = 1            #E 闭包作用域，outer 和 inner 都可用
    def inner():
```

```

    print(count)
    print(ecount)
    lcount = 10    #L 局部作用域，仅限 inner() 函数内可使用
    print(lcount)
inner()

outer()
print(count)

import builtins    #内置作用域是通过名为 builtin 的标准模块实现
dir(builtins)       #可以显示有哪些内置变量，
                    #比如__name__等这些对象就是 B 内置作用域

#=====作用域重叠的问题=====
count = 0          #变量在本层作用域内可修改，在下层作用域不能修改
def outer():
    print(count)
    count = 2      #试图修改上层变量
    ecount = 1

#此时执行 outer() 会报错，抛出 UnboundLocalError 异常
#原因是二：
#1) count = 2 这行语句，是一个定义语句（同时也是赋值语句）
#    表示在函数体内定义了一个新变量 count
#2) 第四行是定义变量，但第三层 print(count) 却在使用变量，这违犯了“变量必须先
赋值再使用”的原则，所以抛出 UnboundLocalError 异常。

#=====变量作用域覆盖=====
#现在修改一下，将 print 语句放在定义之后
count = 0          #变量在本层作用域内可使用和修改
def outer():
    count = 2      #试图修改上层变量，做法是错误
    print(count)

outer()
print(count)      #此时仍然打印 0，而不是 2

#此代码不会抛出异常，但无法得到想要的结果。解释如下：
#1) 第三行，在函数体内定义了一个新变量 count，覆盖了上层的全局的 count 变量
#2) 第四行 print(count) 打印的是局部变量，打印出 2
#3) 第八行 print(count) 打印的是全局变量，打印出 0

```



## 跨域修改变量

正常情况下，不能跨作用域（跨层）修改变量。其根因是，Python 中变量定义语句和赋值语句是合一的，这使得变量修改有可能变成变量定义，进而使得不同作用域下的变量产生混乱。

在某些场景下，如果要跨作用域修改上层的变量，该怎么办？此时，需要使用特殊的关键字来进行变量声明（Python 好奇怪，变量不需要声明即可赋值，这种情况下，却是需要声明的）。

- 1) 如果要修改全局变量，需要使用 `global` 关键字声明
- 2) 如果要修改闭包变量，需要使用 `nonlocal` 关键字声明
- 3) 跨域变量在声明后，才能够被正确修改

```
#如果想在下层/内部作用域修改上层/外部作用域的变量，
#则需要使用 global 和 nonlocal 关键字来显式说明。
num = 1                #全局 num
def fun():
    global num          #显式指定，使用全局变量
    print(num)          #打印 1
    num += 2            #将全局变量修改为 3
    print(num)          #打印 3

fun()
print(num)              #打印 3，用的是全局变量

#=====
#显式
num = 1                 #全局变量
def outer():
    num = 10            #闭包变量
    def inner():
        nonlocal num    #显式指定为本函数外，上层函数内的闭包变量
        print(num)      #打印 10
        num = 100       #修改闭包变量
        print(num)      #打印 100
    inner()
    print(num)          #打印 100, 闭包变量

outer()
print(num)              #打印 1，全局变量
```

```

#=====
spam = '全局变量'
def scope_test():
    def do_local():
        spam = "local spam"           #2. 修改局部变量，不改变闭包变量

    def do_nonlocal():
        nonlocal spam                 #3. 说明下面改变的是闭包变量
        spam = "nonlocal spam"

    def do_global():
        global spam                   #4. 使用全局变量，也不改变闭包变量
        spam = "global spam"

    spam = "test spam"               #1. 闭包变量

    do_local()
    print("After local assignment:", spam)

    do_nonlocal()
    print("After nonlocal assignment:", spam)

    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)      #5. 打印全局变量

#结果是
#After local assignment: test spam
#Before nonlocal assignment: test spam
#After nonlocal assignment: nonlocal spam
#After global assignment: nonlocal spam
#In global scope: global spam

```

## 自定义类

Python 设计之初就是一门面向对象的语言。常说的面向对象编程，指的是编程的三大特性：封装、继承和多态。

- 1) 所谓封装，简单地说就是支持类定义。类，其实是具有相同属性和行为的一组事物的描述，它定义了该事物的公共的属性和行为（方法）的。类的实例化就是具体的单个的类对象。
- 2) 所谓继承，就是允许某个对象类获得另一个类型对象的属性和方法，即一个类（称为子类）可以从另一个类（称为父类或基类）中派生出来。
- 3) 所谓多态，指的是同一个方法在不同的情况中有不同的表现形式，对应的就是：子类可以重载父类方法。换一句话说，同样的方法在不同的子类中有不同的实现方式，这就叫多态。

不过，很遗憾，Python 中的类其实并不完善（被阉割了的类），没有类似 C++，Java 程序中类的 `private` / `public` / `protected` 属性和方法的说法，原则上，Python 中的属性和方法全是 `public` 的。但是，在 Python 也有约定（只一种惯例，并不强制）。

## 类的定义

1. 类的定义格式如下：

```
#基本类定义格式
class 类名:
    类描述体

#继承类定义格式
class 类(基类 1, 基类 2, ...):
    类描述体
```

一般约定，自定义类名的第一个字母要大写，模块名/函数名/方法名的第一个字母小写，以避免搞不清是类还是函数/方法。

## 类成员

2. 每个类，基本有两种类成员：

- a) 类属性，即类中实例化的变量对象。
- b) 类方法，即类中定义的函数。

实例化类对象后，才能够访问类对象的属性和方法。

### 3. 按照约定，类成员（属性和方法）有三种性质：

- a) `public`: 默认类属性和方法都是公有的，可在类的外部（即类对象和派生类）直接访问。
- b) `private`: 在类外部（类对象和派生类）不能直接访问。在 Python 中，对应 `name mangling` (名称重整)，以双下划线 (`__`) 开头的成员，实际上，系统会自动对这个变量前面加上一个前缀，在子类中无法被重载或覆盖。
- c) `protected`: 在类对象中不要直接访问，但自身类和派生类可以直接访问。在 Python 中，以单下划线 (`_`) 开头的变量相当于保护成员，在子类中可以重载或覆盖。

（注：上述只是约定，实际上所有属性和方法都是可以在类外部访问的，参考示例代码）

### 4. 类成员（属性和方法）的引用格式

- a) 在类中，格式为 `self.member`
- b) 在类外，格式为 `obj.member`
- c) 如果是 `name mangling` (名称重整) 的变量，则还要加上类名前缀（如下所示）。

```

class Dog:
    kind = 'dog'
    _name = ''
    __age = 0

    def out(self):
        pStr = 'kind={},name={},age={}'.format(
            self.kind, self._name, self.__age)
        print(pStr)

d1 = Dog()
d1.kind = '哈士奇'
d1._name = '帅帅'
d1._Dog__age = 2
d1.out()

```

## 类属性

5. 类属性，进一步可细分为两种：

a) 类变量

- 1) 在类中直接定义变量并赋值（不需要 self. 前缀）
- 2) 这个变量是所有实例对象共用的。

b) 实例变量

- 1) 在类方法（主要是\_\_init\_\_函数）中赋值的变量，变量前必须加上 self. 前缀。理论上，带有 self. 前缀的所有变量都是实例变量。
- 2) 单个实例对象所特有的。

6. 类变量和实例变量的差别

- 1) 一般情况下，类变量和实例变量的用法是一样的。但在特定情况下，是有区别的。
- 2) 实例变量的修改，只会涉及到单个类对象。
- 3) 但类变量的修改，有两种可能性：

- a) 如果类变量是不可变的数据类型，则只会涉及到单个类对象。
- b) 如果类变量是可变的数据类型，则会涉及到所有的类对象。

```
class people:
    kind = 'Human'           #类变量(不可变)
    clslt = []               #类变量(可变)
    def __init__(self, name):
        self.name = name     #实例变量(不可变)
        self.instlist = []   #实例变量(可变)

    def __str__(self):
        pStr = 'kind={},name={},clslt={},instlist={}'.format(
            self.kind, self.name, self.clslt, self.instlist)
        return pStr
```

```
p1 = people('John')
print(p1.kind, p1.clslt, p1.name)
```

```
p2 = people('Mike')
print(p2)
```

####=====修改实例变量=====

```
p1.name = 'Pone' #修改实例变量 (不可变)
print(p1)
print(p2)
```

```
p1.instlist.append(20) #修改实例变量 (可变)
print(p1)
print(p2)
```

####=====修改类变量=====

```
p1.kind = 'People' #修改类变量 (不可变)
print(p1)
print(p2)
```

```
p1.clslt.append(10) #修改类变量 (可变)
print(p1)
print(p2) #居然p2也变化了!
```

```
class Dog:
    tricks = [] #1. 类属性, 类变量 class variable

    def __init__(self, name):
        self.name = name #2. 类属性, 实例变量 instance variable

    def add_trick(self, trick, name=''): #3. 类方法, 可以操作所有类属性
        self.tricks.append(trick)
        if not name == '':
            self.name = name

d = Dog('Fido')
e = Dog('Buddy')
print(d.name, e.name) #不同实例, 其实例变量不一样
d.add_trick('roll over')
e.add_trick('play dead') #调用不同实例的同一个方法。
print(d.tricks, '\n', e.tricks) #不同实例, 其类变量是相同的
```

#输出结果相同['roll over', 'play dead']

```
dg = Dog('Jack')
dg.add_trick('backward', 'John')
print(dg.name, dg.tricks)
```

## 类方法

### 7. 类方法:

- a) 类方法，和函数定义基本一样，唯一不同的是类方法的第一个参数名称约定是 `self`（这代表类的实例对象）。
- b) 类方法的调用格式有两种：
  - i. `obj.fun()`，即对象.函数
  - ii. `className.fun(obj)`，即类名.函数名(对象)
  - iii. 在调用类方法时不用理会这个参数
- c) 系统规定类有一些特殊的方法，比如 `__init__()`，该方法在类实例化时会自动调用，用于对象的初始化构造，这些系统方法可以被重载，以实现某种特定的功能。

```
class Human:
    #重载系统函数
    def __init__(self, name):
        self.name = name        #1. 定义公共属性
        self.age = 10           #2. 定义公共属性
        self._height = 170      #3. 定义受保护属性，子类可直接使用
        self.__weight = 60      #4. 定义私有属性，子类不可覆盖

    def speak(self):            #5. 定义公共函数
        print('Human speak:name={}, age={}, height={}, weight={}'.format(
            self.name, self.age, self._height, self.__weight))

    def _hu(self):                #6. 定义受保护方法，子类可重载，可覆盖
        print('Human hu:私有函数')
```



```

def __fo(self):
    #7. 定义私有重整方法，子类不可覆盖，系统会自动
    命名（添加前缀）
    print('Human fo:私有重整函数')

#使用类对象
p = Human('John')          #实例化类对象

#访问类属性
print(p.name, p.age)        #1. 直接访问公共属性
print(p._height)            #2. 直接访问私有属性
#print(p.__weight)          #不能访问重整属性, 会抛出 AttributeError 异常
print(p._Human__weight)     #3. 可以这样访问重整私有属性

#调用方法
p.speak()                   #1. 直接调用公共方法
p._hu()                     #2. 直接调用受保护方法
#p.__fo()                   #不能直接调用重整私有方法, 会抛出 AttributeError 异常
常
p._Human__fo()              #3. 可以这样调用重整私有方法

```

## 系统类方法

=====基本访问=====

\_\_new\_\_(): 创建新实例时调用，是一个静态方法

\_\_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

\_\_str\_\_(): 字符串格式函数

\_\_repr\_\_(): 打印，转换

\_\_format\_\_(): 格式化

\_\_hash\_\_(): 哈希操作

\_\_bool\_\_(): 布尔操作

=====属性访问=====

\_\_getattr\_\_():

```

__getattr__():
__setattr__():
__setattribute__():
__delattr__():
__dir__()
===== implementing Descriptors =====
__get__()
__set__()
__delete__()
__set_name__()
=====
__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

```

## 重载运算符

```

#重载运算符（加法+）
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)
```

## 继承类

1. 类还可以从其它基类（父类）进行派生，派生类也叫做子类。父类名称放在子类的括号中
2. 父类可以有多个，但要注意父类的顺序，特别是两个父类有相同的函数名时。在调用子类的方法是，其搜索的顺序如下：子类，第一个基类，第二个基类，...
3. 子类的初始化函数中，强烈建议第一行代码就是要调用父类的初始化函数。因为，大多数类，其属性都是在\_\_init\_\_函数中实例化的，如果不调用基类初始化，有可能在子类就无法直接访问父类的实例属性。
4. 子类要访问父类 的方法，可以有几种方式：
  - a) 使用 super().fun() 格式（建议），明确表示访问的是父类的方法
  - b) 使用 BaseClassName.methodname(self, arguments) 格式（不常用）
  - c) 使用 self.fun() 格式，像使用子类函数一样（但要注意子类是否有同名函数）
5. 子类可以改写/重写父类的方法，此时父类的方法将被覆盖。

6. 跟派生类识别相关的两个内置函数：isinstance(),  
issubclass()。

```
#定义子类
#定义派生类/定义子类
class Student(Human):
    def __init__(self, name, no, grade):
        self.no = no
        self.grade = grade
        self._height = 180          #这个直接对父类变量赋值
        self.__weight = 90          #注：这个不是使用父类的属性，而是新定义了一个子类的实例属性
        super().__init__(name)      #调用父类的初始化方法 （强调建议）

    def talk(self):                  #编写子类的方法
        print('student talk: No={}, Grade={}'.format(self.no, self.grade))
        super()._hu()               #子类也可以直接调用父类的方法
        # self._hu()                #子类可以直接调用父类的方法
        # people._hu(self)           #还可以这样调用，但一定要带上 self 参数，
        # 否则会抛出 TypeError 异常

    def speak(self):               #重载父类的方法
        print('student speak: no={}, age={}'.format(\
            self.no, self.age))

    def __fo(self):
        print('student fo:重整函数')

stu = student('Jonh', 123, '二年级')

#访问类属性
print(stu.name, stu.age, stu._height, stu._Human__weight)  #访问父类属性
print(stu.no, stu.grade, stu._student__weight)             #访问子类属性

stu.talk()           #调用子类公共方法
stu.speak()          #调用子类重载方法

stu._hu()            #调用父类方法
stu._Human__fo()     #调用父类重整方法
stu._student__fo()   #调用子类重整方法

rst = isinstance(stu, student)
rst = isinstance(stu, Human)
```

```
rst = isinstance(student, Human)
rst = isinstance(student, people)
```

#多重继承类定义

```
#class Student(Human, Speaker):
```

#其它类似

## 模块

模块和包是任何大型项目的核心，就连 Python 安装程序本身就是一个包。模块其实是一个文件，其后缀名是.py。

模块中，可包含执行语句(所以模块是可以被执行的)，也可以包含一些声明和定义（比如函数、类、常量等）。

模块可以被别的程序引入，以使用该模块中的函数等功能。

## 导入模块

- 1) 使用 import 导入整个模块，模块只会被导入一次。
- 2) 还可以使用 as 给导入的模块一个别名。
- 3) 调用模块中的函数时采用 module.name.functionname。

#直接导入模块

```
import os
```

```
import numpy as np
```

```
dirs = os.listdir( '/' )          #使用 module.fun 调用
```

## 导入函数

- 4) 使用 `from ...import` 导入指定的函数，也可以导入多个指定的函数。
- 5) 也可以使用 `*` 导入所有的函数（但不建议这样做：有可能导致函数重名，产生意想不到的结果）。
- 6) 这种导入方式，可以直接使用函数名 `fun` 调用，不需要添加模块名前缀。

```
#导入模块中的函数
from os import listdir          #导入指定函数
#from os import listdir, getcwd #导入多个指定的函数
#from os import *               #导入所有函数

dirs = listdir( '/' )           #直接调用函数，不需要前缀 os.
```

## 模块搜索路径

在导入模块时，一般情况下，系统需要知道模块所在的位置。

- 7) 正常情况，Python 会按照 `sys.path` 中的路径搜索模块，其中第一个空格就是当前目录。
- 8) 如果想要导入指定路径的模块，则可以将路径加入 `sys.path` 中，再执行导入语句。
- 9) 也可以采用相对路径来导入模块。

```
import sys
print(sys.path)          #默认路径是环境变量 PYTHONPATH 提供

#添加系统路径
sys.path.append( '/ufs/guido/lib/python' )
#然后就可以导入此目录下的自定义的模块了，这样就不会出错
```

```
import mylog

#相对路径导入模块

from . import mymodule
from .. otherdir import mymodule
```

## *dir()* 函数

可以找到模块内定义的所有名称。

```
import sys

dir()          #列出当前模块定义的所有标识符
dir(sys)       #列出 sys 模块定义的所有标识符
```

## 特殊属性

### `__name__` 属性

每个模块都有一个名字，保存在 `__name__` 变量中。

但这个变量在不同的运行环境下有不同的值，由系统决定。

- 1) 如果模块是在 Python 环境中自己运行，则  
`__name__ = '__main__'`；
- 2) 如果模块是被 import 导入到程序，那么 `__name__` 就是模块名  
(`modulename`)。

所以，此属性可以用来判断一个模块是直接被运行，还是在被另一个程序导入使用。使用如下判断语句，就可以使得这两种运行方式有不同的效果。

```
#mymodule.py
```

```

if __name__ == '__main__':
    #print("表示程序自身在运行") #使用$python mymodule.py 命令执行
    #此处可以放测试代码
else:
    #print("表示本模块被导入") #使用 import mymodule 命令执行
    #此处放一些执行代码

import os
print(os.__name__)    #返回 os

```

## \_\_doc\_\_ 属性

这属性是对模块功能的描述。

## \_\_spec\_\_ 属性

这属性是系统自动生成的对模块位置、调用者的描述。

```

import os
print(os.__doc__)    #返回 os

>>> os.__spec__
ModuleSpec(name='os', loader=<_frozen_importlib_external.SourceFileLoader
object at 0x10e5648d0>,
origin='/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/os.
py')

```

## \_\_all\_\_ 属性

在模块中用来定义可导出的标识符（函数名、类名、常量）等等。

正常情况下，比如使用 `from module import *` 这样的导入语句，会把所有不以下划线开头的符号全部导入。



如果定义了\_\_all\_\_，那么只有显示列出的符号名才会被导出；如果\_\_all\_\_定义为空列表，那么任何符号都不会被导出。

```
#mymodule.py

def spam():
    pass

def grok():
    pass

blah = 42

#可导出的标识符
__all__ = [ 'spam', 'grok' ]
```

## 包

### 基本知识

在 Python 中，模块只是一个文件，那么包则是一个目录。可以认为，包是 Python 模块的一个命名空间。

创建一个包是简单，只要把代码放在相应的目录结构中，并确保每个目录中都定义了一个\_\_init\_\_.py 文件即可。

1. 包相当于一个目录，目录下可以有多个不同的模块。
2. 包目录中必须包含一个\_\_init\_\_.py 的文件，此文件可以是空的，也可以包含一些初始化代码，或者定义变量\_\_all\_\_（该变量指定可以导入本模块的哪些名称）。
3. 包可以嵌套，即包还可以包含子包（类似目录下有子目录）。

```
#package  sound  处理声音文件和数据
sound/                                     顶层包
```

<code>__init__.py</code>	初始化 sound 包
<code>formats/</code>	文件格式转换子包
<code>__init__.py</code>	
<code>wavread.py</code>	
<code>wavwrite.py</code>	
<code>aiffread.py</code>	
<code>aiffwrite.py</code>	
<code>auread.py</code>	
<code>auwrite.py</code>	
...	
<code>effects/</code>	声音效果子包
<code>__init__.py</code>	
<code>echo.py</code>	
<code>surround.py</code>	
<code>reverse.py</code>	
...	
<code>filters/</code>	filters 子包
<code>__init__.py</code>	
<code>equalizer.py</code>	
<code>vocoder.py</code>	
<code>karaoke.py</code>	
...	

## 包的导入

- 无法直接导入包，只能导入包中的模块。在导入模块时，需要指定包前缀（如果子包改名，则所有的导入都要重新修改）。
- 如果要导入同一个包中的另一个子包的模块，也可以使用相对路径来帮助模块导入。

#包的导入格式

```
#import package.subpackage.module
```

#1. 导入包中模块

```
import sound.effects.echo
```

#使用的时候，需要全路径，格式：package.subpackage.module.fun

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

#2. 导入包中特定模块

```
from sound.effects import echo
```

```
echo.echofilter(input, output, delay=0.7, atten=4)    #格式: module.fun
```

#3. 导入模块中特定函数

```
from sound.effects.echo import echofilter
```

```
echofilter(input, output, delay=0.7, atten=4) #可直接使用函数名 fun
```

## 包内引用

#如果包内有子包，子包下有模块，模块要引用另一个模块

#比如，sound.filters.vecoder 模块要使用 sound.effects.echo 模块

#1. 导入时指明包的全路径

```
form soun.effects import echo
```

#2. 相对路径导入

# 当前目录下（同一个包中）的另一个模块

```
from . import equalizer
```

#3. 导入上级模块

```
from .. import formats #上级目录中的包. 注意..后没有斜杠的
```

```
from ..formats import wavwrite
```

## 特殊使用

`__path__`

包提供一个属性`__path__`，里面指定这个包要用到的目录列表。

这个功能不常用，一般用来扩展包里面的模块。

## `__all__`

这个属性和在模块中的定义是一样，用于显示指明本包可以导出哪些标识符。

```
#最简单的__init__.py
#指明本模块可用的标识符

__all__ = ['TestResult', 'TestCase', 'TestSuite',
           'TextTestRunner', 'TestLoader', 'FunctionTestCase', 'main',
           'defaultTestLoader', 'SkipTest', 'skip', 'skipIf', 'skipUnless',
           'expectedFailure', 'TextTestResult', 'installHandler',
           'registerResult', 'removeResult', 'removeHandler']
```

## 六、文件操作

### 基本操作

#### 文件打开

Python 中内置了 `open` 函数(其实就是标准模块 `io`)，用于文件的操作，返回一个 `file` 对象。如果文件不能打开，则会抛出 `OSError` 异常。

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

```
#1. 打开文本文件 Text I/O。参考 TextIOBase
f = open("myfile.txt", "r", encoding="utf-8")
```

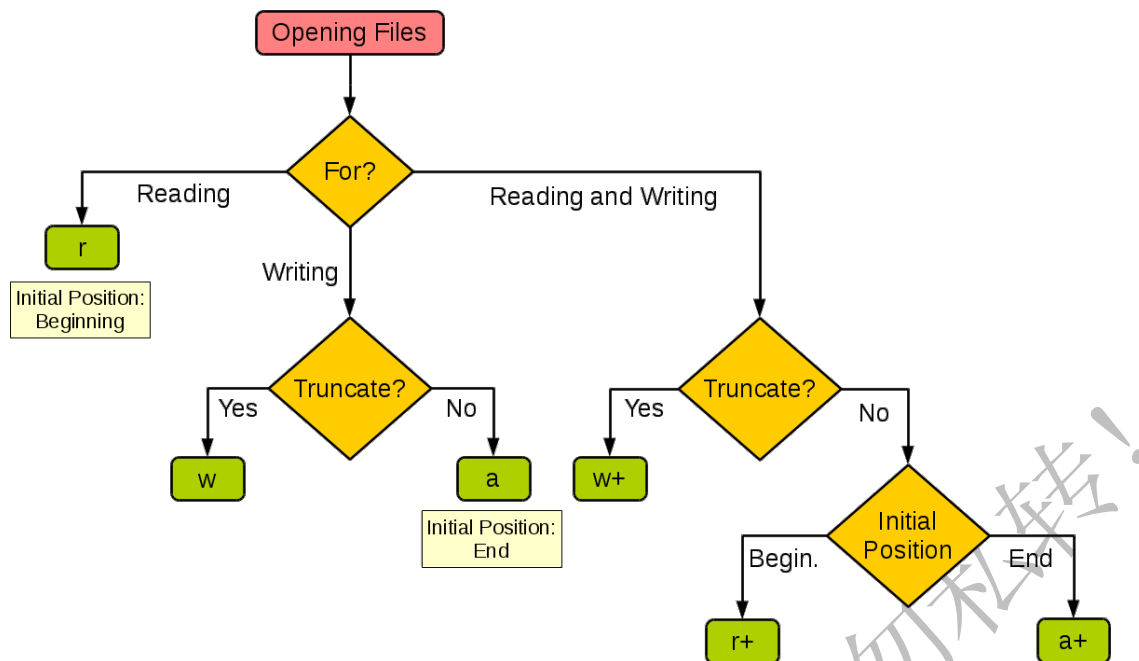
```
#2. 打开二进制文件 Binary I/O 或 buffered I/O  
f = open( 'myfile.jpg' , 'rb' )
```

## 打开模式

mode 的取值如下：

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first 注意：会清空原来文件内容哟！
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)

采用哪种模式依赖于你要对文件有什么样的操作，如下所示：



常见模式：

r 读文本文件；文件不存在时会抛出异常 `FileNotFoundError`；  
 w 写文本，文件不存在就创建文件，如果文件存在则清空文件；  
 wb 读写二进制文件，并清空原内容；  
 r+b 读写二进制文件，但不清空原内容

对文件的操作可以参考 io 模块。

<https://docs.python.org/3/library/io.html#module-io>

## 文本读取

文件打开后就可以对文件内容进行操作了，文本文件常用读取方式：

- 1) 读取一行，返回字符串 `readline()`；
- 2) 读取所有行，返回列表 `readlines()`；
- 3) 读取全部内容 `read()`，返回一个字符串；
- 4) 注：上述所有函数读取的字符串末尾都带有换行符。

#文本文件读取

```

filename = 'text.txt'
f = open(filename, "rt")
if f.readable():
    line = f.readline() #默认读取一行
                        #如果返回空串，表示已经读取到最后一行。

    print(line)
    bt = f.read(5)      #读取接下来的指定的字节数
    print(bt)
    lines = f.readlines() #读取接下来的所有行
    print(lines)

    f.seek(0)          #读取指针回到文件开头 (=f.seek(0, os.SEEK_SET)
    contents = f.read() #一次性读取所有内容
    print(contents)

else:
    print('文件不可读')
f.close()              #关闭文件

```

## 文件写入

向文件写入字符串。

注意：write()/writelines() 不会自动添加换行符，需要手工添加。

```

#文本文件写入
filename = 'text.txt'
f = open(filename, "w") #打开文件写，并清空原有内容

num = f.write("I love programming!\n") #返回写入的字符数
num = f.write("I love game!\n")        #注：write 不会自动添加换行符
                                        #如果你要写入一行，则需要自己手工添加。

f.close()

```

## 文件关闭

1. 文件打开操作结束后，必须关闭文件；否则有可能造成文件锁死，其它的程序无法访问文件。
2. 关闭文件使用 `close()` 函数。
3. 但是，如果文件打开后，在读或写的过程中发现异常，有可能会跳过 `close` 函数，导致文件得不到正确关闭；如果想要文件得到正确关闭，建议使用上下文语句 `with` 关键字，当 `with` 代码块结束（不管是正常还是异常），来自动正常关闭文件。

```
#文件无法正常关闭
import io
filename = 'text.txt'
f = open(filename, "rt")
f.seek(1, io.SEEK_END) #读取指针在文件末尾后，会抛出异常
                        #实际上 SEEK_END=0, SEEK_CUR=1, SEEK_END=2
line = f.readline()
f.close()              #异常后的语句都不会被执行，文件没有正常关闭

#文件在 with 语句外，系统会自动关闭文件
filename = 'text.txt'
with open(filename, "rt") as f:
    lines = f.readlines()

#文件在 with 语句外，会自动关闭文件，不必要显式誉
for line in lines():
    print(line.rstrip())
```

## 关于换行符

在 Python 中，官方默认的换行符就是 `\n`。但如果要读写文本文件中的换行符，不同的系统，换行符也不一样。

在 Unix 系统中，采用 `\n` 表示 (newline)；在 Windows 中，采用 `\r\n` 表示；还有的操作系统有可能使用 `\r` 表示换行符。



在 open 函数打开文件时，newline 参数有特殊的含义：

1) 读文件时，

- a) 默认 newline=None，表示启用“通用型换行符模式”，系统会自动识别所有换行符\r, \n, \r\n，并会将其转化成\n并返回给调用者；
- b) 如果 newline=''（空串），依然会启用“通用型换行符模式”，自动识别换行符，但不会转换字符，而是直接传给调用者，由调用者来处理。
- c) 如果 newline 为其它字符，系统也不会处理换行符，只是直接将带换行符的字符串传给调用者。

2) 写文件时，

- a) 默认 newline= None，字符串中的换行符\n 会被自动转换为当前系统默认的换行符（os.linesep）。
- b) 如果 newline='' 或者\n，则在写入文件时不会发生转换。
- c) 如果 newline 为其它字符，则会把\n 转换为给定的字符写入文件。

```
# wintxt.txt 是在 Windows 下的纯文本文件
# 默认 newline=None
filename = 'wintxt.txt'
with open(filename, 'r') as f:
    cnts = f.read()          #返回的换行符被转换为\n
    print(cnts)

with open(filename, 'r', newline='') as f:
    cnts = f.read()          #返回的换行符是\r\n, 没有被转换
    print(cnts)
```

## 常见文件异常

OSError: 最基本的异常

FileNotFoundError: 文件不存在

FileExistsError:文件已经存在

EOFError: 文件读末尾错误

## 二进制文件操作

二进制文件的读取对象一般是字节串。所以，常用的函数是 `read()`, `write()`。

### 基本读写

```
filename = 'mybinary.dat'
bts1 = bytearray(range(10))
bts2 = bytearray(b'I love you!')
with open(filename, 'wb') as f:
    f.write(bts1)
    f.write(bts2)

with open(filename, 'rb') as f:
    bt1 = f.read(10)
    bt2 = f.read()
    pass
```

### 随机读取

对于二进制文件，经常使用 `seek` 来调整读取指针的位置，从而实现随机访问文件内容。

函数原型：

`seek(offset[, whence])` 函数可以改变读取指针的位置

# `offset` 表示指定位置的偏移量，可以是任何整数（正或负或零）

# `whence` 指针的位置，有三个取值：

# `SEEK_SET` or 0 - 默认是首部，`offset` 必须为 0 或正整数

# `SEEK_CUR` or 1 - 当前位置，`offset` 为任何整数

# `SEEK_END` or 2 - 文件末尾，`offset` 必须为 0 或负整数

```
##文件读取指针的管理=====
```

```

#

import io

filename = 'mybinary.dat'
with open(filename, 'rb') as f:
    if f.seekable():
        #判断是否支持随机访问
        bts = f.read(5)           #读取指定的 5 个字符
        offset = f.tell()        #返回当前的位置，字节数
        f.seek(0)                #默认返回文件首部
        bts = f.read(4)
        f.seek(2, io.SEEK_SET)    #返回文件首部后的 2 个字节。
        line = f.read(6)         #读取指定的 6 个字符
        f.seek(-4, io.SEEK_CUR)   #回退 4 个字节。
        contexts = f.read()       #读取余下的所有字符串，此时指针在末尾
        f.seek(-6, io.SEEK_END)   #指针跑到文件末尾倒数第 6 个字节。
        chars = f.read()
        pass
    else:
        print(' 文件不支持随机访问! ')

```

## 文件对象（略）

内置的 open 函数返回的是一个文件对象，Python 的 io 模块中提供了三类文件对象：

- 1) text files
- 2) raw binary files
- 3) buffered binary files

## 对象序列化

文本文件只能保存字符串，二进制文本又无法区分字节串含义。

如果要保存对象，并且在读取的时候以对象的方式读取并正确处理。在这种场景下，就可以使用数据序列化和反序列化模块 pickle 模块。

```
import pickle

data1 = { 'a' :[1,2],
          'b' : ' this is a string' ,
          'c' :None}
data2 = [1,2,3]

#将对象写入文件（序列化）
filename = 'data.pkl'
with open(filename, "wb" ) as f:
    pickle.dump(data1,f)
    pickle.dump(data2,f)

#将文件内容读出到对象（反序列化）
with open(filename, "rb" ) as f:
    dt1 = pk.load(f)
    dt2 = pk.load(f)
```

## JSON 文件操作

JSON（JavaScript Object Notation），是一种轻量级的数据交换格式，采用完全独立于编程语言的文本格式来存储和表示数据。它采用键值对来保存对象（" firstName" : " John" ）。

其读写方式和序列化相同（略）。

```
import json
nums = [1,2,5,0]
filename = 'numbers.json'

#以 JSON 格式保存对象数据
with open(filename, 'w') as f:
    json.dump(nums, f)

#读取 JSON 格式的对象数据
with open(filename) as f:
```

```
nums = json.load(f)
print(nums)
```

## INI 文件操作

大多数的配置文件为 INI 文件，可以使用 configparser 模块进行读取和写入。

<https://docs.python.org/3/library/configparser.html>

## 写配置文件

```
#写配置文件.ini
import configparser
config = configparser.ConfigParser()

config['DEFAULT'] = {'ServerAliveInterval': '45',
                    'Compression': 'yes',
                    'CompressionLevel': '9'}
config['DEFAULT']['ForwardX11'] = 'yes'

config['topsecret.server.com'] = {}
topsecret = config['topsecret.server.com']
topsecret['Port'] = '50022'
topsecret['ForwardX11'] = 'no'

with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

于是生成如下 `example.ini` 配置文件

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes
```

```
[bitbucket.org]
```

```
User = hg
```

```
[topsecret.server.com]
```

```
Port = 50022
```

```
ForwardX11 = no
```

## 读配置文件

```
# 读取配置文件
config = configparser.ConfigParser()
# >>> config.sections()
config.read('example.ini')

sects = config.sections()
topsecret = config['topsecret.server.com']
sFrd = topsecret['ForwardX11']
port = topsecret['Port']

for key, val in config['DEFAULT'].items():
    print(key, '=', val)
```

## CSV 文件操作

CSV (Comma-Separated Value) 文件，是一种特殊的文本文件，每一行数据用逗号隔开，通常用于电子表格软件和纯文本之间交互数据。理论上，CSV 文件可以当成纯文本一样处理，但是如果文本中本来就包含有逗号，就比较麻烦。而 CSV 模块可以较好地处理字段值中本身就带有逗号的情况。

csv 模块提供了两种读写对象：

- 1) csv.writer, csv.reader
- 2) csv.DictWriter, csv.DictReader

在打开 csv 文件时，要求带 `newline=""` 参数。

## 序列读取

```
import csv

##列表方式写入
#
filename = 'test.csv'
with open(filename, 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['firstname', 'lastname'])
    writer.writerow(['John', 'Alex'])
    writer.writerow(['Mike', 'Michal'])

##列表方式读取
with open(filename, 'r', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
    print('读取行数=', reader.line_num)

##元组方式写入
#
DATA = (
    (9, 'Web Clients', 'base64,urllib'),
    (10, 'Web Programming:CGI&WSGI', 'cgi,time,wsgiref'),
    (13, 'Web Services', 'urllib, twython')
)

filename = 'mycsv2.csv'
with open(filename, 'w', newline='') as f:
    writer = csv.writer(f)
    for record in DATA:
        writer.writerow(record)

with open(filename, 'r', newline='') as f:
    reader = csv.reader(f)
    for id, tl, des in reader:
        print(id, tl, des)
```

## 字典读写

```
##字典方式写入
#
filename = 'test2.csv'
fieldName = ['名字', '姓']
with open(filename, 'w', newline='') as f:
    writer = csv.DictWriter(f, fieldnames=fieldName)

    writer.writeheader()    #写标题
    writer.writerow({'名字': '小明', '姓': '黄'})
    writer.writerow({'名字': '大刚', '姓': '李'})
    writer.writerow({'名字': '国安', '姓': '张'})

##字典方式读取
#
# 注意：第一行为字典的键，其余行为字典的值
with open(filename, 'r', newline='') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row['名字'], row['姓'])    #注意，键必须与第一行标题相同
```

## Excel 文件操作

- 1) 读写 excel 文件，需要安装 xlrd 和 xlwt 这两个库，可以处理 xls 文件（最大行数 65535）。
- 2) 如果要处理更大行数的，则需要 openpyxl 库，可处理 xlsx/xlsm 文件（最大行业 1048576）。

详细参考 <https://openpyxl.readthedocs.io/en/stable/>

- 3) 也可以使用 Pandas 模块来读取 Excel 文件。

详细参考后续大数据分析专题。



## 七、异常操作

异常指的是程序在运行过程中发生错误，比如除零错误等，导致程序中止。异常不像语法错误，语法错误在程序初期就可以检测出来，但异常却在真实的应用场景才可能被抛出和发现。

## 异常处理

- 1、 异常处理完整的语句：try-except-else-finally，其中 try 和 except 子句是必须的，except 子句可以重复多次，else 和 finally 子句都是可选的。
- 2、 异常子句执行的顺序有两种：
  - a) 无异常时，执行顺序：try 子句→else 子句→finally 子句。
  - b) 有异常时，执行顺序：try 子句→except 子句 →finally 子句。不管有无异常，finally 子句（如果有）总是会被执行的。
- 3、 如果在执行 try 子句中发生异常，那么 try 子句余下的部分将被忽略，转而执行相应的 except 子句。
- 4、 如果没有异常与任何 except 匹配，那么这个异常将会传递给上层的 try 中；如果上层没有 try，程序将崩溃并中止。
- 5、 except 子句可以同时捕获多个异常并处理
- 6、 可以使用 raise 语句重新抛出异常，或者抛出指定的异常。

```
import sys

filename = 'test.txt'
try:
    f = open(filename, 'r')
    line = f.readline()
    # num = int(line.strip())
    # raise Exception('手工抛出的异常!')      #手工抛出异常
```

```

except FileNotFoundError as err:                                #对单个异常处理
    print("文件不存在! 错误代码={}, 错误描述={}, 文件名={}".format( \
        err.errno, err.strerror, err.filename))
except (ValueError, TypeError) as err:                        #匹配多个异常
    print("类型或值转换错误! except={}".format(err))
except:                                                         #匹配所有错误
    print("未知错误!", sys.exc_info())
    #raise                                                       #也可以继续抛出异常
else:
    f.close()

```

## 异常的顺序

- 7、 如果有多个 except 子句，程序将会按照位置的先后顺序进行匹配，所以，需要注意异常的层次性。
- 8、 如果异常类之间存在派生关系，要求子类在前，父类在后（异常的层次性请参考下面小节）；否则，子类永远不会被匹配。

#下面示例将优先匹配父类 OSError 异常，而不是 FileNotFoundError 子类

```

filename = 'myfile.txt'
try:
    f = open(filename, 'r')
    line = f.readline()
except OSError as err:
    print("文件操作失败! error={}".format(err))
except FileNotFoundError as err:
    print("文件不存在! 错误代码={}, 错误描述={}, 文件名={}".format( \
        err.errno, err.strerror, err.filename))

```

## 异常时环境

当程序发生异常时，异常会保存在系统中，可以通过 `sys.exc_info()` 函数返回 3 元组（异常类，异常对象，`traceback` 对象）。

## 异常类层次

Python 异常类层次如下：

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |   +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- OSError
        |   +-- BlockingIOError
        |   +-- ChildProcessError
        |   +-- ConnectionError
        |       |   +-- BrokenPipeError
        |       |   +-- ConnectionAbortedError
        |       |   +-- ConnectionRefusedError
        |       |   +-- ConnectionResetError
        |   +-- FileExistsError
```

```

|      +--- FileNotFoundError
|      +--- InterruptedError
|      +--- IsADirectoryError
|      +--- NotADirectoryError
|      +--- PermissionError
|      +--- ProcessLookupError
|      +--- TimeoutError
+--- ReferenceError
+--- RuntimeError
|      +--- NotImplementedError
|      +--- RecursionError
+--- SyntaxError
|      +--- IndentationError
|      +--- TabError
+--- SystemError
+--- TypeError
+--- ValueError
|      +--- UnicodeError
|      +--- UnicodeDecodeError
|      +--- UnicodeEncodeError
|      +--- UnicodeTranslateError
+--- Warning
|      +--- DeprecationWarning
|      +--- PendingDeprecationWarning
|      +--- RuntimeWarning
|      +--- SyntaxWarning
|      +--- UserWarning
|      +--- FutureWarning
|      +--- ImportWarning
|      +--- UnicodeWarning
|      +--- BytesWarning
|      +--- ResourceWarning

```

## 自定义异常

- 9、 自定义异常一般从 Exception 继承，大多数异常类名都以 Error 结尾。
- 10、 可以根据需要再次抛出异常。

# 1. 异常一般从 Exception 派生

# 2. 大多数异常类名都以 Error 结尾

```
class MyError(Exception):
    def __init__(self, code, msg=""):
        self.code = code
        self.msg = msg
    def __str__(self): #重载类的系统方法
        return repr('errCode={},errMsg={}'.format(self.code, self.msg))

##
import os
try:
    logFile = 'Log.txt'
    if not os.path.exists(logFile):
        raise MyError(2, '文件不存在')
except MyError as err:
    print(err)
```

## 八、并发操作

<https://docs.python.org/3/library/threading.html>

## 进程和线程

### 进程

进程是一个执行中的程序，每个进程都拥有自己的地址空间、内存、数据栈以及其他用于跟踪执行的辅助数据。操作系统会管理所有进程的执行，并为进程合理地分配时间。

程序中也是可以使用 fork 或 spawn 来创建新的进程的。

### 线程

线程是由进程启动的，一个进程中的各个线程与主线程共享同一个数据空间。所以，线程间通信要比进程间通信容易得多。

Thread 类对象表示的线程有三个要素：name（线程名），ident（线程 ID），daemon（是否守护线程）。

默认情况下，在函数执行完成后，线程就自动退出，也可以调用 `thread.ext()` 函数主动退出线程。

相比而言，多线程比较适合于 IO 密集型的应用，能够提升并发性能；如果是计算密集型操作，由于 Python 的 GIL 的限制，多线程并没有太多用处，此时应该使用多进程，而不是多线程，以便让 CPU 的其他内核来执行。

## 守护线程

守护线程一般是一个等待客户端请求服务的服务器。如果把一个线程设置为守护线程，就表示这个线程是不重要的，进程退出时不需要等待这个线程执行完成。

要将一个线程设置为守护线程，需要在启动线程之前执行赋值语句：`thread.daemon=True`，也可以用这个属性来判断线程的守护状态。另外，一个新的子线程会继承父线程的守护标记。

正常情况下，Python 进程（主线程）将在所有非守护线程退出之后才退出。

## 多线程并发

创建线程的三种方式：

- 1) 创建 Thread 的实例，传给它一个函数
- 2) 派生 Thread 的子类，并创建子类的实例
- 3) 创建 Thread 的实例，传给它一个可调用的类实例（不常用）

## 传入函数

把函数当成一个线程的执行功能。

```
import threading

#####1. 传入函数，启动线程#####fib
##
def fib(n):
    # sleep(0.005)
    if n < 2:
        return 1
    return (fib(n-2)+fib(n-1))

def loop(nsec):
    print('开始线程, name={}, id={}, daemon={}, time={}'.format(
        threading.current_thread().name,
        threading.current_thread().ident,
        threading.current_thread().daemon, ctime()))
    sleep(nsec)
    print('结束线程: id={}'.format(threading.current_thread().ident))

print('开始=====')
thrdloop = threading.Thread(target = loop, args = (4,))
thrdfib = threading.Thread(target = fib, args = (20,))

thrdloop.start()

print('开始线程, name={}, id={}, daemon={}, time={}'.format(
    thrdfib.name, thrdfib.ident, thrdfib.daemon, ctime()))
thrdfib.start()
thrdfib.join()
print('结束线程: id={}'.format(thrdfib.ident))

thrdloop.join()

print('结束=====')
```

## 派生线程类

- 1、 从 `threading.Thread` 类派生自己的线程类 (`MyThread`)
- 2、 记得初始化时要首先调用基类的初始化化
- 3、 重载基类的 `run()` 函数，实现线程的功能，系统会自动调用。
- 4、 使用 `start()` 函数启动线程
- 5、 使用 `join()` 函数等等线程退出

```
##=====2. 派生线程类，启动线程=====
##

class MyThread(threading.Thread):
    def __init__(self, nsec):
        threading.Thread.__init__(self) #记得调用基类的初始化
        self.nsec = nsec

    def run(self):
        print(' 开始线程,name={}, id={}, daemon={}, time={}'.format(
            self.name, self.ident, self.daemon, ctime()))
        sleep(self.nsec)
        print(' 结束线程: {},time={}'.format(self.ident, ctime()))

print('\n 开始=====')
thrdOne = MyThread(4)
thrdTwo = MyThread(2)

thrdOne.start()
thrdTwo.start()

thrdOne.join()
thrdTwo.join()

print(' 结束=====')
```



## 线程类 *Thread*

Thread 对象属性	
name	线程名
ident	线程 ID
daemon	布尔值，是否守护线程
Thread 对象方法	
start	启动线程
run()	可重载的自定义线程功能
join(timeout=None)	等待线程结束，阻塞调用
is_alive()	判断线程是否存活
Thread 其它函数	
active_count()	当前活动的线程个数
current_thread()	返回当前线程对象 Thread
enumerate()	当前活动的线程对象列表
settrace(func)	为所有线程设置一个 trace 函数
setprofile(func)	为所有线程设置一个 profile 函数
stack_size(size=0)	返回/设置新建线程的栈大小

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
```

```
def run(self):
    f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
    f.write(self.infile)
    f.close()
    print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

## 线程同步机制

多线程提升了并发的性能，并同时带来新的问题。当多个线程访问相同的数据资源时，有可能会产生资源冲突和不一致性。

解决冲突的办法就是同步机制，经常使用的同步原语有：锁/互斥，信号量，以及 Queue 对象。

### 锁 Lock

锁是所有机制中最简单的，最低级的机制。

锁有两种状态：锁定和未锁定。锁的操作只有两个：获得锁和释放锁。

多线程争夺锁时，允许第一个获得锁的线程进入临界区，并执行代码，所有之后到达的线程将被阻塞，直到第一个线程操作完成，退出临界区，并释放锁。此时，其他等待的线程可以获得锁并进入临界区。即锁可以保证在同一时刻只有一个线程在访问资源。

- 1) 当使用 `acquire()` 时，表示锁定资源，阻塞其它线程访问；
- 2) 当使用 `release()` 时，表示释放资源，允许其它线程访问。

- 3) 注意：一定别忘记 `release()` 资源，否则资源锁死，造成系统全部阻塞。因此，建议使用 `with` 语句，这样可避免由于异常导致跳过 `release()` 代码而锁死资源。

```
##=====3. 锁同步=====
##
print('[]'*20)
from random import randrange
from threading import Thread, currentThread, Lock
from time import sleep, ctime

loops = []
for i in range(5):
    loops.append(randrange(2, 10))

remaining = set()
lock = Lock()

def loop(nsec):
    myName = currentThread().name

    lock.acquire()                #1) 使用锁同步
    remaining.add(myName)
    print('{} 开始线程 {}'.format(ctime(), myName))
    lock.release()

    sleep(nsec)


    # lock.acquire()
    with lock:                    #2) 建议使用上下文管理锁，即使中间出异常，
也能正常释放锁
        remaining.remove(myName)
        print('{} 结束线程 {}, {} 秒'.format(ctime(), myName, nsec))
        print('剩下: {}'.format(remaining or 'None'))
    # lock.release()

for pause in loops:
    thrd = Thread(target=loop, args=(pause,))
    thrd.start()
print('OK')
```

## 信号量 *BoundedSemaphore*

信号量用于多线程竞争有限资源的情况。

- 1) 信号量的本质是一个计数器；
- 2) 当调用 `acquire()` 时表示资源消耗，计数器递减，但不能小于 0；
- 3) 当调用 `release()` 时表示函数资源，计数器递增，但不能超过最大资源数。



```
#假定只有 5 个槽，可放 5 颗糖
##=====4. 信号量同步=====
##
print('=='*20)
from random import randrange
from threading import Thread, currentThread, BoundedSemaphore
from time import sleep, ctime

lock = Lock()
MAX = 5
candyTray = BoundedSemaphore(MAX)

def refill():
    lock.acquire()
    print('放糖')
    try:
        candyTray.release()
    except ValueError:
        print('槽满了')
    else:
        print('OK')
    lock.release()

def buy():
    lock.acquire()
    print('购买')
    if candyTray.acquire(False):
        print('OK')
    else:
        print('空了')
    lock.release()

def producer(loops):
```

```

    for i in range(loops):
        refill()
        sleep(randrange(3))

def consumer(loops):
    for i in range(loops):
        buy()
        sleep(randrange(3))

nloops = randrange(2, 6)
Thread(target=consumer, args=(randrange(nloops, nloops+MAX+2),)).start()

Thread(target=producer, args=(nloops,)).start()

print("OK")

```

## 线程安全队列 *Queue*

在信号量中，信号量的处理并不是阻塞的，资源没有了只是抛出异常。如果想阻塞调用，则可以使用队列 `queue` 模块中的队列进行线程间通信。

`Queue` 类提供了一个线程安全的队列，其内部已经实现并发安全的保护处理，不需要其它附加的同步机制。

queue 模块中的类	
<code>Queue(maxsize=0)</code>	先进先出队列。如果设定最大值，则在队列没有空间时阻塞，否则（没有指定最大值）为无限队列
<code>LifoQueue(maxsize=0)</code>	后进先出队列。其余同上
<code>PriorityQueue(maxsize=0)</code>	优先级队列。其余同上
Queue 对象方法	
<code>qsize()</code>	返回队列大小（不确定）

<code>empty()</code>	判断是否为空
<code>full()</code>	判断是否已满
<code>put(item, block=True, timeout=None)</code>	将 item 放入队列，block 表示如果队列有可用空间前是否阻塞调用；如果 block 为 False，则在没有空间时就会抛出 Full 异常
<code>put_nowait(item)</code>	同上
<code>get(block=True, timeout=None)</code>	返回元素。如果 block=True，则一直阻塞到有可用的元素为止；有可能抛出 Empty 异常。
<code>get_nowait()</code>	和 get(False) 相同
<code>task_done()</code>	用于表示队列中某个元素已执行完成，该方法会被下面的 join() 使用。
<code>join()</code>	在队列中所有元素执行完毕并调用上面的 task_done() 之前，保持阻塞。

```
##=====5. 队列同步=====
```

```
##
```

```
from random import randint
from time import sleep
from queue import Queue
from threading import Thread
```

```
def writeQ(queue):
    print('生产元素')
    queue.put('xxx', True)
    print('队列大小={}'.format(queue.qsize()))
```

```
def readQ(queue):
```

```

    val = queue.get(True)
    print('取出元素 "{}"，队列大小={}'.format(val, queue.qsize()))

def writer(q, nloops):
    for i in range(nloops):
        writeQ(q)
        sleep(randint(1, 3))

def reader(q, nloops):
    for i in range(nloops):
        readQ(q)
        sleep(randint(2, 5))

nloops = randint(2, 5)
q = Queue(32)

thrdWt = Thread(target=writer, args=(q, nloops))
thrdRd = Thread(target=reader, args=(q, nloops))

thrdWt.start()
thrdRd.start()

thrdWt.join()
thrdRd.join()

```

## 多进程(略)

前面所述，如果是计算密集型程序，应该使用多进程，以便能够更好地利用多核 CPU 来处理。常用的多进程模块有几个：

**subprocess** 模块：可以单纯地执行任务，或者通过标准文件 (stdin, stdout, stderr) 进行进程间通信。

**multiprocessing** 模块：允许为多核或多 CPU 派生进程，其接口与 threading 模块相似。该模块同样包括在共享任务的进程间传输数据的多种方式。

**concurrent.futures** 模块：新的高级库，只在“任务”级别进行操作。

九、文档化与测试自动化

文档化

作为一门开发语言，良好的文档说明是必须的。而 Python 专门设计了文档、注解等机制。系统会自动解析你编写的模块、类和函数等，并可以在代码中查看这些说明文档。

模块说明\_\_doc\_\_

你现在编写了一个模块（一个.py 文件），你想告诉别人这个模块的主要功能。

对模块的描述内容，需要满足如下规范：

- 1) 描述内容必须放在.py 文件中的所有 Python 语句的前面。
- 2) 描述内容必须以多行注释号(三个'''或三个"")括起来。
- 3) 可以使用\_\_doc\_\_属性查看某模块的说明。

其实，任何模块都有如下的系统属性：

系统属性	描述（除__doc__外，其余都是系统自动生成的）
__name__	模块名称，不同的场景下，模块名称也不同： 1) 如果模块是在 Python 中直接运行，则__name__是”__main__”。 2) 如果模块是被导入，则其__name__就是模块的名字（即文件名称）。



<code>__doc__</code>	模块的描述信息（用户按规则编写）
<code>__loader__</code>	本模块的调用者信息
<code>__file__</code>	模块所在的全路径
<code>__package__</code>	模块所在的包
<code>__cached__</code>	模块生成机器码文件.pyc 的全路径
<code>__spec__</code>	模块的规范说明，包括 name, loader, origin(即 file)信息。

```
import survey
```

```
##显示模块描述(主要有 7 个)
```

```
print('1 模块 name: ', survey.__name__)      #模块的名称
print('2 模块 doc: ', survey.__doc__)         #模块的描述信息
print('3 模块 loader:', survey.__loader__)    #模块调用者
print('4 模块 file:', survey.__file__)        #模块所在全路径
print('5 模块 package:', survey.__package__)  #模块所在包
print('6 模块 debug', survey.__cached__)      #模块的机器码文件.pyc 的全路径
print('7 模块 debug', survey.__spec__)        #模块说明，包括三个信息：
                                              #name, loader, origin(即 file)
```

## 函数功能说明 `__doc__`

写一个函数，有时需要描述此函数的功能，以便其他人在调用。

对函数功能的描述内容，需要满足如下规范：

- 4) 描述内容必须放在函数定义后的第一行。
- 5) 描述内容必须以多行注释号(三个'''或三个"")括起来。
- 6) 可以使用 `__doc__` 属性查看该函数的说明。
- 7) 如果没有说明，则 `__doc__` 为 None。

```
def openLogFile():
    '''\
```

```
本函数 openLogFile 负责打开日志文件，以便记录系统运行的信息
请注意在使用请关闭日志文件。
'''
```

```
pass
```

```
#查看函数功能描述
```

```
print(openLogFile.__doc__)
```

## 函数参数说明\_\_annotations\_\_

函数参数的注解（Annotations）用于说明参数类型等等，以方便调用者知道传入合适的数据类型。

- 8) 形参描述内置在函数内部
- 9) 使用:来说明形参的数据类型
- 10) 使用->说明返回值的数据类型
- 11) 系统会自动生成参数注解保存在\_\_annotations\_\_属性中。
- 12) 如果没有说明，则\_\_annotations\_\_为空字典{}

```
def outLogInfo(info: str, Lev: int = 2) -> bool:
    pass
    return True
```

```
#查看函数参数说明
```

```
>>>outLogInfo.__annotations__
{'info': <class 'str'>, 'Lev': <class 'int'>, 'return': <class 'bool'>}
```

## 类说明\_\_doc\_\_

如果要对自定义类进行说明，类似函数说明，放在类定义的第一行，用多行注释号括起来，系统生成的描述放在\_\_doc\_\_属性中；类方法的描述和前面函数的描述完全一样，放在\_\_annotations\_\_属性中。

```
class MyBase():
    """\
        本类为自定义类，用于其它类的基类。
    """
    kind = 'people'

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

    def outLogInfo(self, nLev:int, info:str)->bool:
        """\
            本方法 fun 实现记录日志的功能。
        """
        pass
        return True

##=====类及类方法的描述内容=====
##
print(survey.MyBase.__doc__)
print(survey.MyBase.outLogInfo.__doc__)
print(survey.MyBase.outLogInfo.__annotations__)
```

## 类说明\_\_annotations\_\_

同函数说明。

## 代码测试

在项目开发时，必须要对代码进行测试。

1. Python 提供 unittest 模块进行单元测试。
2. 测试类必须派生于 unittest.TestCase 类。
3. 测试类中可定义测试函数，函数必须以 Test\_ 开头。
4. 一般一个测试用例对应一个测试函数。
5. 测试类中，使用断言来判断执行结果是否符合预测。
6. 测试文件最后，执行 unittest.main() 函数。

详细参考测试用例类的函数：

<https://docs.python.org/3/library/unittest.html#unittest.TestCase>

## 测试函数

假定模块 survey.py 中有如下函数：

```
#Filename: survey.py

#函数
def getFormatedName(first , last, middle=' '):
    if middle:
        fullName = first + ' ' + middle + ' ' + last
    else:
        fullName = first + ' ' + last
    return fullName.title()
```

现在对这个函数做自动化单元测试，则需要生成一个测试类 MyFunTest。测试代码全部在 TestSruvey.py 文件中。

```
#Filename: TestSurvey.py

import unittest          #导入单元测试库
import survey            #导入被测试的类、函数等模块

class MyFunTest(unittest.TestCase):
    #用例 1
```

```

def test_getFormattedName1(self):
    fullName = survey.getFormattedName( 'jack' , 'wang' )
    self.assertEqual( fullName, 'Jack Wang')

#用例 2
def test_getFormattedName2(self):
    fullName = survey.getFormattedName( 'jack' , 'wang' ,
    'mozart' )

    self.assertEqual( fullName, 'Jack Mozart Wang')

#用例 3 (检查空格, middle 是空格)

def test_getFormattedName2(self):
    fullName = survey.getFormattedName( 'jack' , 'wang' ,
    ' ' )

    self.assertEqual( fullName, 'Jack Wang' )

unittest.main() #文件末尾不要少了这句

```

=====

现在运行 TestSurvey.py 脚本, 就会看到 3 个用例, 有 1 个失败。

```

=====
FAIL: test_getFormattedName3 (__main__.MyFunTest)
-----
Traceback (most recent call last):
  File "/Users/fusx/Documents/OneDrive/Python/语言基础/TestSurvey.py", line
36, in test_getFormattedName3
    self.assertEqual(fullName,'Jack Wang')
AssertionError: 'Jack  Wang' != 'Jack Wang'
- Jack  Wang
?      --
+ Jack  Wang

-----
Ran 3 tests in 0.019s

FAILED (failures=1)

```

接下来, 就是检测原始的代码, 修改代码, 使得测试用例通过。比如, 将 `if middle:` 修改为 `if middle.strip():`

## 测试类

类的测试和函数测试基本类似。

一般地，在测试类的测试函数，是相互独立的。但在测试类中，有一个 `setUp()` 函数，只会在开始时调用一次，然后再调用 `test_` 函数。所以一般类对象的创建可以放在这里，以供所有的测试函数使用。

```
#Filename: survey.py

#类:匿名调查
class AnonymousSurvey():
    def __init__(self, question):
        self.question = question
        self.responses = []

    def showQuestion(self):
        print(self.question)

    def storeResponse(self, newResponse):
        self.responses.append(newResponse)

    def showResults(self):
        print('Survey Result:')
        for res in self.responses:
            print('-'+res)
```

现在在测试模块中添加测试类 `MyClassTest`，以及测试用例。

```
#Filename: TestSurvey.py

#测试类
class MyClassTest(unittest.TestCase):

    #用例 1
    def test_Response(self):
        #构建临时对象测试
        question = "你学的第一种语言是什么?"
        mySurvey = survey.AnonymousSurvey(question)
```

```
mySurvey.storeResponse('中文')
self.assertIn('中文', mySurvey.responses)

#此函数只在开始时调用一次，所以对象构建可以放在这里
def setUp(self):
    question = "你还学了哪些语言?"
    self.mySurvey = survey.AnonymousSurvey(question)
    self.responses = ['English', 'Spanish', 'Mandarin']

#用例 2，可以使用测试类的公共的对象 self.mySurvey
def test_singleResponse(self):
    self.mySurvey.storeResponse(self.responses[0])
    self.assertIn(self.responses[0], self.mySurvey.responses)

#用例 3
def test_MultiResponse(self):
    for res in self.responses:
        self.mySurvey.storeResponse(res)

    for res in self.responses:
        self.assertIn(res, self.mySurvey.responses)

unittest.main()
```

附：常用标准库

参考文档链接：

<https://docs.python.org/3/library/index.html>

标准库	描述
os	提供与操作系统的接口，目录/文件访问相关的函数
os.path	提供系统的路径相关函数
shutil	提供最常用 的文件和目录管理任务，比如复制、移动文件等
glob	使用通配符搜索文件列表
sys	命令行参数，输出重定向，程序终止
re	字符串正则匹配
math	数学运算，浮点运算
statistics	基本统计模块（平均值、中位数、方差等）
random	随机数生成和选择
urllib/urllib.request	互联网及网络通信协议
smtplib	邮箱访问协议
zlib, gzip, bz2, lzma, zipfile, tarfile	数据压缩和解压
doctest, unittest	质量控制，单元测试。
datetime, date, time, timedelta	提供时间处理相关功能
timeit	性能评估，还有 profile, pstats 模块中了也有对时间的测量
local	访问本地文化 culture 相关的数据格式，比如使用的货币符号，数字是否用逗号分隔等
struct	用于封包和解包
threading	线程操作



```

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

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

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

#如果在命令行运行 python demo.py o two three, 且在 demo.py 中可以访问命令行参数:
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']

#sys 模块中有标准属性 stdin, stdout, stderr
#其中 stderr 常用于强制输出信息 (即使 stdout 被重定向)
>>> sys.stderr.write('警告, 日志没有发现')

#终止脚本最直接的方式就是调用 sys.exit()

>>> import re
>>> re.findall(r'\b[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'

#math 可访问 C 库的浮点数功能
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)

```

10.0

```
>>> 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

>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095

>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         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()

>>> 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

#性能评估
>>> 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

#自动嵌入测试
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() # automatically validate the embedded tests

```