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

Ancaonda 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）

1. 然后，设置脚本文件的执行权限。

sudo chmod a+x hello.py

1. 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。

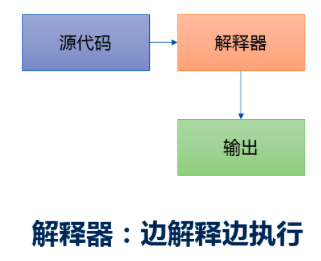
./hello.py

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。



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。

1. 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。

1. 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。

1. 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。

1. 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。

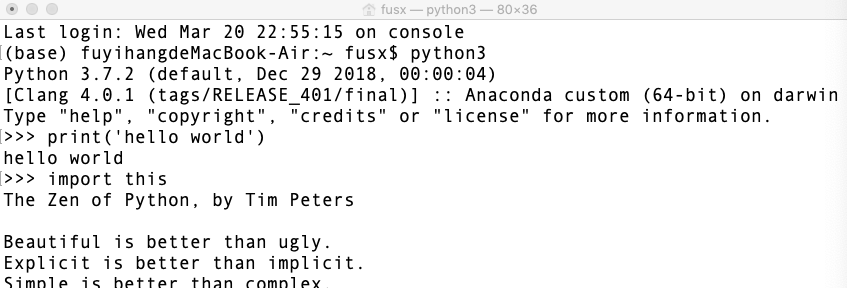
VSCode：微软最好的开发工具，同样功能强大，关键还免费。

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。



### 2）用Jupyter Notebook交互

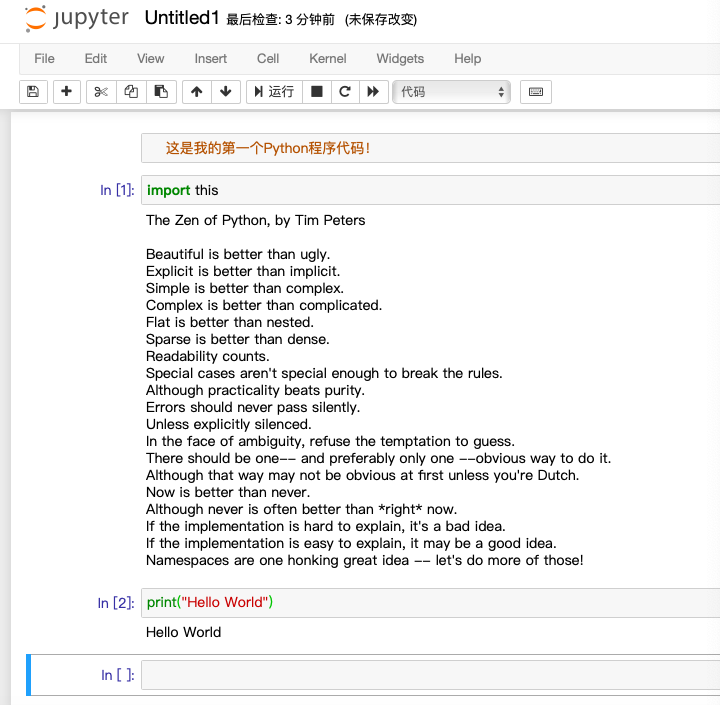
较多的开发者，选择Jupyter Notebook开发工具和环境，它是基于IPython解释器的一个GUI交互开发界面，基于Web浏览器的开发环境 ，方便项目文件的管理。它支持注释/代码混写，支持多行代码编写等等。

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。

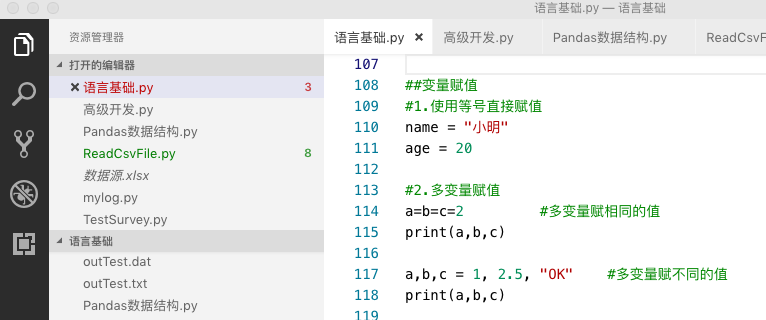
现在，运行Anaconda Navigator，你可以看到jupyter Notebook，点击运行（Launch），然后就会打开一个Web浏览器，点击右上角的New->Python3就会建一个Untitled1文件，于是就进入Jupter Notebook交互界面。

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 Sompackage 升级包

>pip uninstall Sompackage 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：



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并不完全是后向兼容的，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。为了解决定个版本不兼容的问题，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 mypy2env.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：输入冒号会可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](https://cloud.tencent.com/developer/doc/1198腾讯云)

## 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：
   1. 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。
   2. 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：
   1. 对于模块名，一般是全部小写字母；
   2. 对于内置的标识符(函数、类)，都全部是小写字母；
   3. 对于自定义类名，一般首字母大写（驼峰式）；
   4. 对于类成员，一般首字母小写（驼峰式或下划线分隔）
   5. 对于变量名，一般首字母小写（驼峰式或下划线分隔）；
   6. 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：

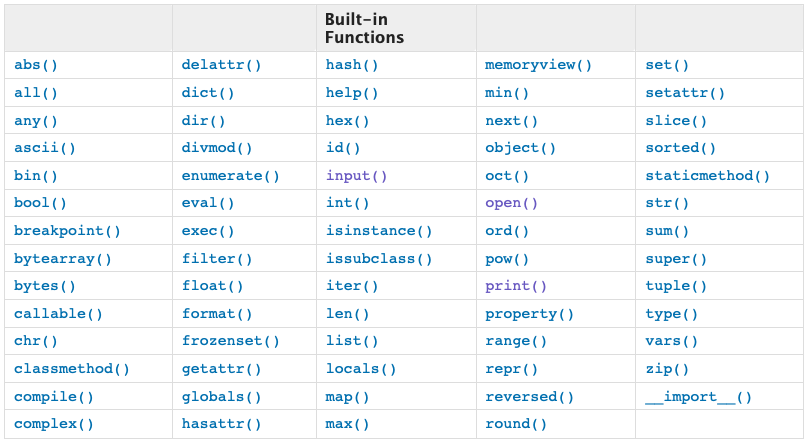
##### 内置特殊对象

* 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

##### 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>。



##### 内置标准数据类型

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

##### 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。

##### 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

##### 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

##### 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，声明为整数型后，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](#_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. 采用字符器类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%操行符，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。

1. 如果括号中没有数字，则参数要严格按位置顺序对应；
2. 括号中也可以指定数字，则数字用于指向参数的位置顺序。
3. 如果括号中指定关键字参数，则按照关键字匹配。
4. 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 | 黃色

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。

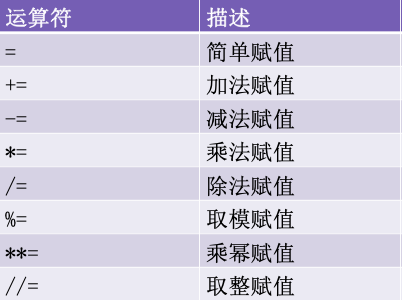
### 其它特殊语句

其它特殊语句，可以单独使用，也有可能会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)



a += b，相当于a = a+b，其余类似

### 2.算术运算符(8个)



### 3.位运算符(6个)



### 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）。



### 5.布尔运算符(3个)



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)



**>>>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。



**>>>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@ |
| +,- | 加、减 |
| <<, >> | 位移 |
| **&** | **按位与** |
| ^ | 按位取反 |
| | | 按位或 |
| in, not in, is, is not, <, <=, >, >=, !=, == | 比较/成员/身份运算符 |
| not x | 布尔非 |
| and | 布尔与 |
| or | 布尔或 |
| if-else | 条件表达式 |
| lambda | 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
   1. 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。
   2. 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
   1. 相当于变量或对象保存的数据类型。
   2. 可用内置函数type()来查看对象的数据类型。
3. 对象value
   1. 对象value就是对象真实的值。
   2. 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）
   1. 对象alias（别名）就是变量名。变量名其实就是对象的引用名称。
   2. 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）。
   3. 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) #内存中创建了一个整数对象3，但没有引用名字**

**4467530992**

**>>> id(4) #内存中创建另一个整数对象4**

**4467531024**

**>>> id(5) #ID号在增加**

**4467531056**

**>>>**

**>>> n = 3 #现在变量赋值为3，相当于给对象一个名称叫n**

**#也相当于变量指向或引用了内存中的对象3**

**>>> id(n) #发现变量n的ID与对象3是一样的。**

**4467530992**

**>>>n2 = n**

**>>> id(n) == id(n2))**

**True #说明两个变量指向的是同一个对象**

**#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：

1. 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。
2. type()函数，用来打印出对象的数据类型。
3. 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 #n2变量也指向同一个对象**

**>>> n is n2 #用is操作符检验这两个对象是不是同一个**

**True**

**>>>id(n) == id(n2)**

**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。



#### 变量指向变化

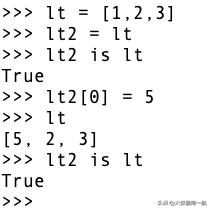
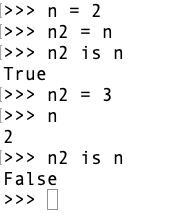
变量修改（变量赋值）时，变量的指向是否会发生变化，是会受到变量的数据类型的影响的。

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。**

### 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** |
| --- | --- | --- |
| d | 格式化整数 |  |
| b | 格式化二进制 |  |
| o | 格式化八进制 | (1) |
| x | 格式化十六进制（小写） | (2) |
| X | 格式化十六进制（大写） | (2) |
| e | 浮点指数形式（小写）-科学计数法 | (3) |
| E | 浮点指数形式（大写）-科学计数法 | (3) |
| 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ascii()转换) |  |
| !s | 字符串(用[str()](https://docs.python.org/3/library/stdtypes.html#str)转换) |  |
| !r | 字符串(用repr[()](https://docs.python.org/3/library/stdtypes.html#str)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. |
| '+' | 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π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 )** |
| abs(x) | 绝对值，如abs(-10) 返回 10 |
| ceil(x) | 向上取整数，如math.ceil(4.1) 返回 5 |
| sqrt(x) | 返回数字x的平方根。 |
| exp(x) | 返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045 |
| fabs(x) | 返回数字的绝对值，如math.fabs(-10) 返回10.0 |
| floor(x) | 返回数字的下舍整数，如math.floor(4.9)返回 4 |
| log(x) | 如math.log(math.e)返回1.0,math.log(100,10)返回2.0 |
| log10(x) | 返回以10为基数的x的对数，如math.log10(100)返回 2.0 |
| max(x1, x2,...) | 返回给定参数的最大值，参数可以为序列。 |
| min(x1, x2,...) | 返回给定参数的最小值，参数可以为序列。 |
| modf(x) | 返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。 |
| pow(x, y) | x\*\*y 运算后的值。 |
| round(x [,n]) | 返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。 |

##### 随机数函数

需要导入random库

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

##### 三角函数

需要导入math库。

|  |  |
| --- | --- |
| **函数** | **描述** |
| acos(x) | 返回x的反余弦弧度值。 |
| asin(x) | 返回x的反正弦弧度值。 |
| atan(x) | 返回x的反正切弧度值。 |
| atan2(y, x) | 返回给定的 X 及 Y 坐标值的反正切值。 |
| cos(x) | 返回x的弧度的余弦值。 |
| hypot(x, y) | 返回欧几里德范数 sqrt(x\*x + y\*y)。 |
| sin(x) | 返回的x弧度的正弦值。 |
| tan(x) | 返回x弧度的正切值。 |
| degrees(x) | 将弧度转换为角度,如degrees(math.pi/2) ， 返回90.0 |
| [radians(x)](http://www.runoob.com/python3/python3-func-number-radians.html) | 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

1. 反斜杠'\'可以用来转义特殊字符，比如\n表示换行。
2. 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。



#### 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

1. 字符串为不可变数据类型，不能够修改其中的部分字符。
2. 字符串为序列变量，可以遍历访问。
3. 也可以使用下标索引访问，下标有两种方式：
   1. 从左往右以 0 开始
   2. 从右往左以 -1 开始
4. 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 | capitalize() 将字符串的第一个字符转换为大写 |
| 2 | center(width, fillchar=‘ ’)返回一个指定宽度 width 居中的字符串，fillchar 为填充的字符，默认为空格。 |
| 3 | count(str, beg= 0,end=len(string))返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数 |
| 4 | bytes.decode(encoding="utf-8", errors="strict")  Python3 中没有 decode 方法，但我们可以使用 bytes 对象的 decode() 方法来解码给定的 bytes 对象，这个 bytes 对象可以由 str.encode() 来编码返回。 |
| 5 | encode(encoding='UTF-8',errors='strict')  以 encoding 指定的编码格式编码字符串，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace' |
| 6 | endswith(suffix, beg=0, end=len(string)) 检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False. |
| 7 | expandtabs(tabsize=8)  把字符串 string 中的 tab 符号转为空格，默认的空格数是 8 。 |
| 8 | find(str, beg=0 end=len(string))  检测 str 是否包含在字符串中，如果指定范围 beg 和 end ，则检查是否包含在指定范围内，如果包含返回开始的索引值，否则返回-1 |
| 9 | index(str, beg=0, end=len(string))  跟find()方法一样，只不过如果str不在字符串中会报一个异常. |
| 10 | isalnum()  如果字符串至少有一个字符并且所有字符都是字母或数字则返 回 True,否则返回 False |
| 11 | isalpha()  如果字符串至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False |
| 12 | isdigit()  如果字符串只包含数字则返回 True 否则返回 False.. |
| 13 | islower()  如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False |
| 14 | isnumeric()  如果字符串中只包含数字字符，则返回 True，否则返回 False |
| 15 | isspace()  如果字符串中只包含空白，则返回 True，否则返回 False. |
| 16 | istitle()  如果字符串是标题化的(见 title())则返回 True，否则返回 False |
| 17 | isupper()  如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False |
| 18 | join(seq)  以指定字符串作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串 |
| 19 | len(string)  返回字符串长度 |
| 20 | ljust(width[, fillchar])  返回一个原字符串左对齐,并使用 fillchar 填充至长度 width 的新字符串，fillchar 默认为空格。 |
| 21 | lower()  转换字符串中所有大写字符为小写. |
| 22 | lstrip()  截掉字符串左边的空格或指定字符。 |
| 23 | maketrans()  创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。 |
| 24 | max(str)  返回字符串 str 中最大的字母。 |
| 25 | min(str)  返回字符串 str 中最小的字母。 |
| 26 | replace(old, new [, max])  把 将字符串中的 str1 替换成 str2,如果 max 指定，则替换不超过 max 次。 |
| 27 | rfind(str, beg=0,end=len(string))  类似于 find()函数，不过是从右边开始查找. |
| 28 | rindex( str, beg=0, end=len(string))  类似于 index()，不过是从右边开始. |
| 29 | rjust(width,[, fillchar])  返回一个原字符串右对齐,并使用fillchar(默认空格)填充至长度 width 的新字符串 |
| 30 | rstrip()  删除字符串字符串末尾的空格. |
| 31 | split(str="", num=string.count(str))  num=string.count(str)) 以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num+1 个子字符串 |
| 32 | splitlines([keepends])  按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。 |
| 33 | startswith(substr, beg=0,end=len(string))  检查字符串是否是以指定子字符串 substr 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查。 |
| 34 | strip([chars])  在字符串上执行 lstrip()和 rstrip() |
| 35 | swapcase()  将字符串中大写转换为小写，小写转换为大写 |
| 36 | title()  返回"标题化"的字符串,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle()) |
| 37 | translate(table, deletechars="")  根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 deletechars 参数中 |
| 38 | upper()  转换字符串中的小写字母为大写 |
| 39 | zfill (width)  返回长度为 width 的字符串，原字符串右对齐，前面填充0 |
| 40 | isdecimal()  检查字符串是否只包含十进制字符，如果是返回 true，否则返回 false。 |

### 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。
   1. 从左往右以 0 开始
   2. 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

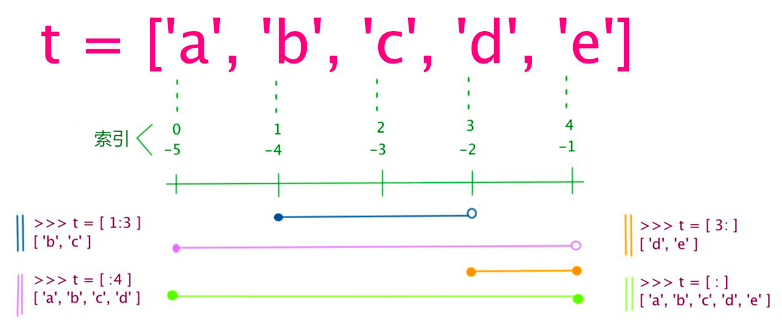
#### 列表切片

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>=0且i<= 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’]

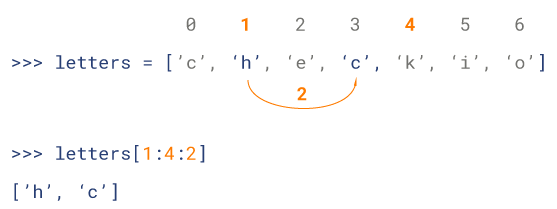
newlt = lt[2:] #返回[’c’,’d’,’e’],默认尾下标到结尾

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]



#### 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 | list.insert(index, obj)将对象插入列表 |
| 6 | list.pop([index=-1])移除列表中的一个元素（默认最后一个元素），并且返回该元素的值 |
| 7 | list.remove(obj)移除列表中某个值的第一个匹配项 |
| 8 | list.reverse()反向列表中元素 |
| 9 | list.sort(cmp=None, key=None, reverse=False)对原列表进行排序 |
| 10 | list.clear()清空列表 |
| 11 | list.copy()复制列表 |

### 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]) #1.直接定义**

**tp = () #2.空元组**

**tp = (20,) #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

1. 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。
2. 元组也可以使用遍历语句。
3. 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

1. 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。
2. 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。

**>>>lt = [1,2,3]**

**>>>tp = (‘ab’, 12, lt)**

**>>>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, lt) #变量重新赋值**

**>>>tp[2][0] = 4 #修改元素列表中的值**

**>>>print(tp)**

**('ab', 34, [4, 2, 3])**

### 5.字典dictonary

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

字典和列表类似，两者区别在于：

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

#### 字典定义

1. 字典用{}标识，用逗号分隔元素，用:分隔键值（键-值）。
2. 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：

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

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

### 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. int(x [,base]) | 将x转换为一个整数 |
| 1. float(x) | 将x转换到一个浮点数 |
| 1. complex(real [,imag]) | 创建一个复数 |
| 1. str(x) | 将对象 x 转换为字符串 |
| 1. repr(x) | 将对象 x 转换为表达式字符串 |
| 1. eval(str) | 用来执行表达式，并返回表达式的值 |
| 1. tuple(s) | 将序列 s 转换为一个元组 |
| 1. list(s) | 将序列 s 转换为一个列表 |
| 1. set(s) | 转换为可变集合 |
| 1. dict(d) | 创建一个字典。d 必须是一个序列 (key,value)元组。 |
| 1. frozenset(s) | 转换为不可变集合 |
| 1. chr(x) | 将一个整数转换为一个字符 |
| 1. ord(x) | 将一个字符转换为它的整数值 |
| 1. hex(x) | 将一个整数转换为一个十六进制字符串 |
| 1. oct(x) | 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, timedelt。

#### 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timedelt

timedelt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。ragne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. start <= x <stop
2. x = start + n\* 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**

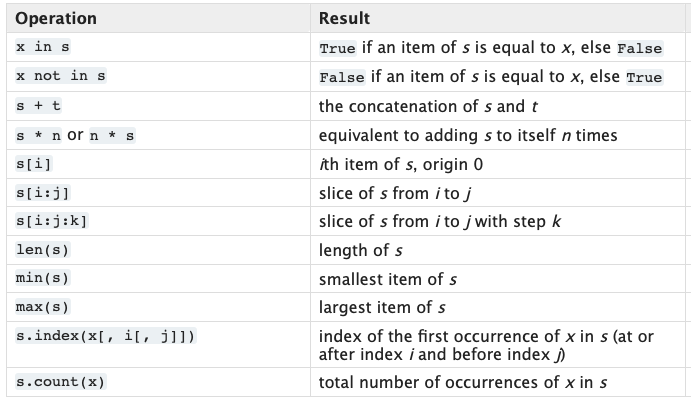
**print('==end==')**

#### 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：

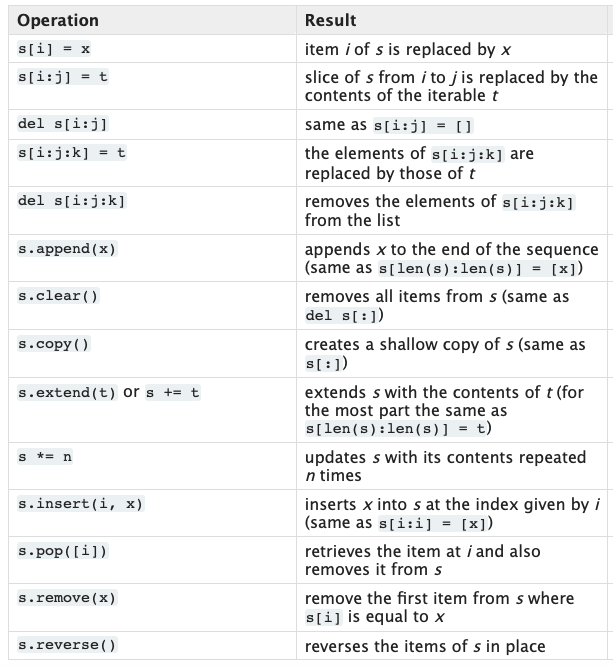
1. 不可变序列类：str,range,tupel, bytes
2. 可变序列类：list, bytearray, memoryview

序列类有一些共同的特征，所有序列类支持如下操作：



**>>>** "gg" **in** "eggs"

True

可改变的序列支持如下操作：  


#### 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。

1. \_\_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。

1. 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

1. 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:**

**curson.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

1. 函数定义时可以有形参，也可以不带形参。
2. 形参可以是必须的，也可以是可选形参（即有默认值）。
3. 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(): #1. 无参数，无返回。相当于return None

print(“Hello World!”)

#return None

def hello2(msg): #2.有参数，无返回

print(msg)

return

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

1. 函数参数可以没有，也可以有，甚至可以有多个形参。
2. 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）。
3. 如果参数有默认值，其位置应该靠后。
4. 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。
5. 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() #打印一个换行

return

fun1(10,30,’45’) #输入多个参数

def fun2( \*\*var): #2.形参为可变参数，即字典

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’) #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：

1. 通过return返回数据，可以返回多个。
2. 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): #形参为可变参数

mylist.append([1,2,3])

return

ls = [10,20,30] #7.通过可变列表传递

print(“函数调用前列表：”, ls) # 调用前和后的列表是同一个

fun(ls)

print(“函数调用后列表:”,ls)

#### 函数调用

1. 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。
2. 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。
3. 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）。
4. 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='voom'):

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

1. 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。
2. 也可以给匿名函数取一个函数名。
3. 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

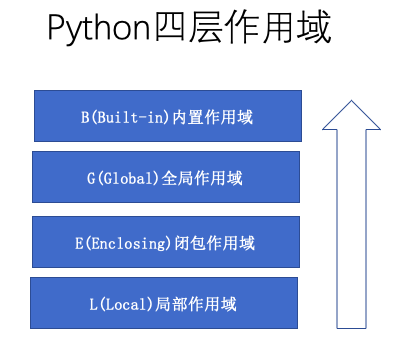
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

#### 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：



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

1. 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？*）

1. 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。

1. 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都可使用**

**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/外部作用或的变量，**

**#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**

**#Befor 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 /protectd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

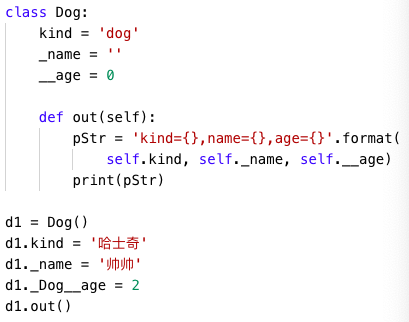
1. 每个类，基本有两种类成员：
   1. 类属性，即类中实例化的变量对象。
   2. 类方法，即类中定义的函数。

实例化类对象后，才能够访问类对象的属性和方法。

1. 按照约定，类成员（属性和方法）有三种性质：
   1. public：默认的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。
   2. 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。
   3. pretected: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）

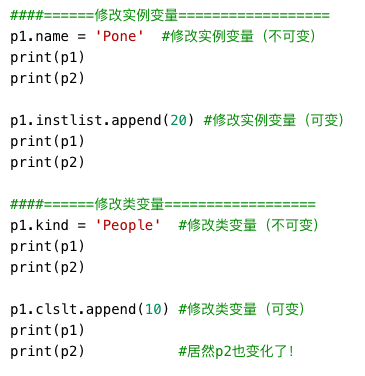
1. 类成员（属性和方法）的引用格式
   1. 在类中，格式为self.member
   2. 在类外，格式为obj.member
   3. 如果是name mangling(名称重整)的变量，则还要加上类名前缀（如下所示）。



#### 类属性

1. 类属性，进一步可细分为两种：
   1. 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**。
   2. 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。**
2. 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：
          1. 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。
          2. 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 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 varible**

**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

1. 类方法：
   1. 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）。
   2. 类方法的调用格式有两种：
      1. obj.fun()，即对象.函数
      2. className.fun(obj)，即类名.函数名(对象)
      3. 在调用类方法时不用理会这个参数
   3. 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**

**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\_\_():

\_\_getattriibute\_\_():

\_\_setattr\_\_():

\_\_setattribute\_\_():

\_\_delatrr\_\_():

\_\_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：
   1. 使用super().fun()格式（建议），明确表示访问的是父类的方法
   2. 使用BaseClassName.methodname(self, arguments)格式（不常用）
   3. 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 = issubclass(student, Human)**

**rst = issubclass(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modulename.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

1. 使用from …import导入指定的函数，也可以导入多个指定的函数。
2. 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）。
3. 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，系统需要知识模块所在的位置。

1. 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。
2. 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。
3. 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

\_\_init\_\_.py 初始化 sound 包

formats/ 文件格式转换子包

\_\_init\_\_.py

wavread.py

wavwrite.py

aiffread.py

aiffwrite.py

auread.py

auwrite.py

...

effects/ 声音效果子包

\_\_init\_\_.py

echo.py

surround.py

reverse.py

...

filters/ filters 子包

\_\_init\_\_.py

equalizer.py

vocoder.py

karaoke.py

...

#### 包的导入

1. 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）。
2. 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bufferd I/O

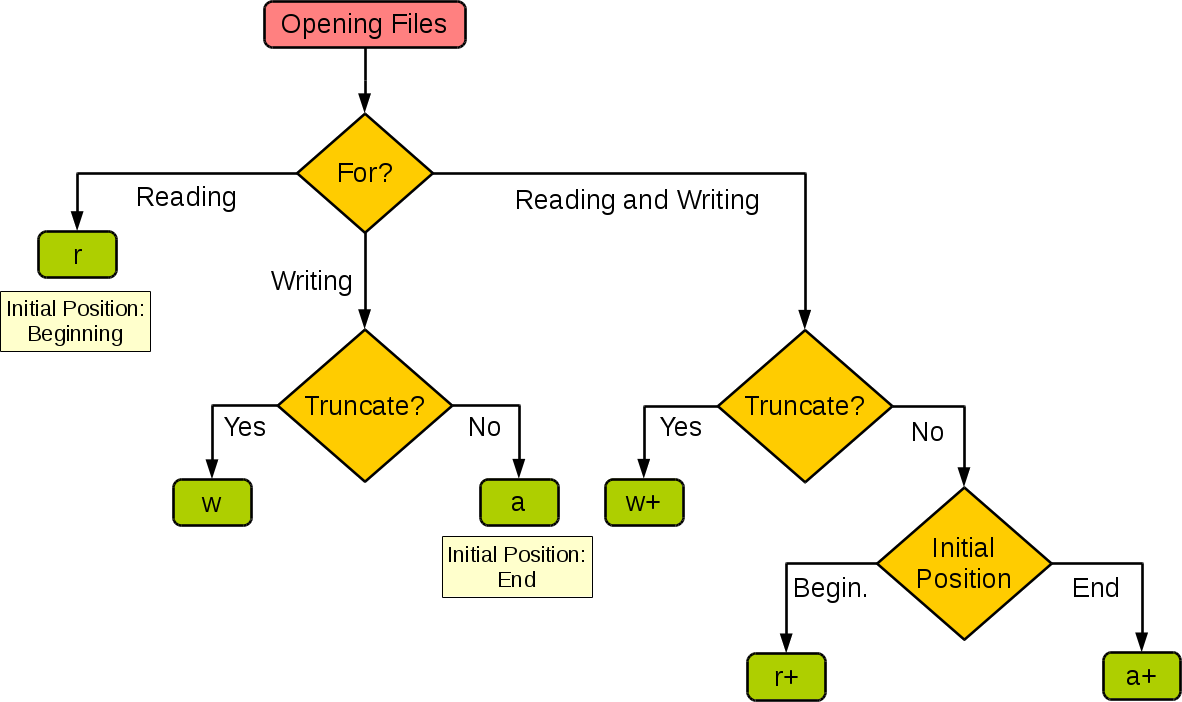
f = open(‘myfile.jpg’, ‘rb’)

#### 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，有可以会跳过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，不必要显式誊

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，
   1. 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；
   2. 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。
   3. 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，
   1. 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）。
   2. 如果newline=’’或者\n，则在写入文件时不会发生转换。
   3. 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个节节。

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 binarry 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')

scts = 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/>

1. 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，excep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：
   1. 无异常时，执行顺序：try子句🡪else子句🡪finally子句。
   2. 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。

1. 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。
2. 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。
3. except子句可以同时捕获多个异常并处理
4. 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

1. 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。
2. 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

1. 自定义异常一般从Exception继承，大多数异常类名都以Error结尾。
2. 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，有可能会导致资源冲突和不一致性。

解决冲突的办法就是同步机制，经常用的同步原语有：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，也能够正常释放锁

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** |  |
| Queue(maxsize=0) | 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 |
| LifoQueue(maxsize=0) | 后进先出队列。其余同上 |
| PriorityQueue(maxsize=0) | 优先级队列。其余同上 |
| **Queue对象方法** |  |
| qsize() | 返回队列大小（不确定） |
| empty() | 判断是否为空 |
| full() | 判断是否已满 |
| put(item,block=True,timeout=None) | 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 |
| put\_nowait（item） | 同上 |
| get(block=True,timeout=None) | 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。 |
| get\_nowait() | 和get(False)相同 |
| task\_done() | 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。 |
| join() | 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。

### Python的GIL

## 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）。 |
| \_\_doc\_\_ | 模块的描述信息（用户按规则编写） |
| \_\_loader\_\_ | 本模块的调用者信息 |
| \_\_file\_\_ | 模块所在的全路径 |
| \_\_package\_\_ | 模块所在的包 |
| \_\_cached\_\_ | 模块生成机器码文件.pyc的全路径 |
| \_\_spec\_\_ | 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：

1. 描述内容必须放在函数定义后的第一行。
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\_\_属性查看该函数的说明。
4. 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。

1. 形参描述内置在函数内部
2. 使用:来说明形参的数据类型
3. 使用->说明返回值的数据类型
4. 系统会自动生成参数注解保存在\_\_annotations\_\_属性中。
5. 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\_getFormatedName1(self):

fullName = survey.getFormatedName(‘jack’, ‘wang’)

self.assertEqual(fullName, ‘Jack Wang)

#用例2

def test\_getFormatedName2(self):

fullName = survey.getFormatedName(‘jack’, ‘wang’, ‘mozart’)

self.assertEqual(fullName, ‘Jack Mozart Wang)

#用例3 (检查空格，middle是空格)

def test\_getFormatedName2(self):

fullName = survey.getFormatedName(‘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\_getFormatedName3 (\_\_main\_\_.MyFunTest)

----------------------------------------------------------

Traceback (most recent call last):

File "/Users/fusx/Documents/OneDrive/Python/语言基础/TestSurvey.py", line 36, in test\_getFormatedName3

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, zipfie, tarfile | 数据压缩和解压 |
| doctest，unittest | 质量控制，单元测试。 |
| datetime, date, time,timedelt | 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'\bf[a-z]\*', 'which foot or hand fell fastest')

['foot', 'fell', 'fastest']

>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')

'cat in the hat'

#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