## 第一章Python 简介

**Python**（英国发音：/ˈpaɪθən/ 美国发音：/ˈpaɪθɑːn/），是一种广泛使用的高级编程语言，属于通用型编程语言，由吉多·范罗苏姆创造，第一版发布于1991年。作为一种解释型语言，Python的设计哲学强调代码的可读性和简洁的语法（尤其是使用空格缩进划分代码块，而非使用大括号或者关键词）。相比于C++或Java，Python让开发者能够用更少的代码表达想法。不管是小型还是大型程序，该语言都试图让程序的结构清晰明了。

与Scheme、Ruby、Perl、Tcl等动态类型编程语言一样，Python拥有动态类型系统和垃圾回收功能，能够自动管理内存使用，并且支持多种编程范式，包括面向对象、命令式、函数式和过程式编程。其本身拥有一个巨大而广泛的标准库。

Python 解释器本身几乎可以在所有的操作系统中运行。Python的正式直译器CPython是用C语言编写的、是一个由社群驱动的自由软件，目前由Python软件基金会管理。

Python的创始人为吉多·范罗苏姆。1989年的圣诞节期间，吉多·范罗苏姆为了在阿姆斯特丹打发时间，决心开发一个新的脚本解释程式，作为ABC语言的一种继承。之所以选中Python作为程式的名字，是因为他是BBC电视剧——蒙提·派森的飞行马戏团的爱好者。ABC是由吉多参加设计的一种教学语言。就吉多本人看来，ABC这种语言非常优美和强大，是专门为非专业程式设计师设计的。但是ABC语言并没有成功，究其原因，吉多认为是非开放造成的。吉多决心在Python中避免这一错误，并取得了非常好的效果，完美结合了C和其他一些语言。

就这样，Python在吉多手中诞生了。实际上，第一个实现是在Mac机上。可以说，Python是从ABC发展起来，主要受到了Modula-3（另一种相当优美且强大的语言，为小型团体所设计的）的影响。并且结合了Unix shell和C的习惯。

吉多是Python的主要开发者，决定整个Python语言的发展方向。Python社群经常称呼他是终身仁慈独裁者（BDFL），2018年7月12日，他宣布不再担任Python社区的BDFL。

Python 2.0于2000年10月16日发布，增加了实现完整的垃圾回收，并且支持Unicode。同时，整个开发过程更加透明，社群对开发进度的影响逐渐扩大。

Python 3.0于2008年12月3日发布，此版不完全兼容之前的Python原始码。不过，很多新特性后来也被移植到旧的Python 2.6/2.7版本。

**标准库:**

Python拥有一个强大的标准库。Python语言的核心只包含数字、字符串、列表、字典、文件等常见类型和函数，而由Python标准库提供了系统管理、网络通信、文本处理、数据库接口、图形系统、XML处理等额外的功能。

Python标准库的主要功能有：

1. 文本处理，包含文本格式化、正则表达式匹配、文本差异计算与合并、Unicode支持，二进制数据处理等功能
2. 文件处理，包含文件操作、创建临时文件、文件压缩与归档、操作配置文件等功能
3. 操作系统功能，包含线程与进程支持、IO复用、日期与时间处理、调用系统函数、日志（logging）等功能
4. 网络通信，包含网络套接字，SSL加密通信、异步网络通信等功能
5. 网络协议，支持HTTP，FTP，SMTP，POP，IMAP，NNTP，XMLRPC等多种网络协议，并提供了编写网络服务器的框架
6. W3C格式支持，包含HTML，SGML，XML的处理。
7. 其它功能，包括国际化支持、数学运算、HASH、Tkinter等

Python社区提供了大量的第三方模块，使用方式与标准库类似。它们的功能覆盖科学计算、Web开发、数据库接口、图形系统多个领域。第三方模块可以使用Python或者C语言编写。SWIG,SIP常用于将C语言编写的程序库转化为Python模块。Boost C++ Libraries包含了一组函式库，Boost.Python，使得以Python或C++编写的程式能互相调用。Python常被用做其他语言与工具之间的“胶水”语言。

**第三方库：**

* Web框架

Django

开源Web开发框架，它鼓励快速开发,并遵循MVC设计，开发周期短。

Flask

轻量级的Web框架。

Pyramid

轻量，同时有可以规模化的Web框架，Pylon projects 的一部分。

ActiveGrid

企业级的Web2.0解决方案。

Karrigell

简单的Web框架，自身包含了Web服务，py脚本引擎和纯python的数据库PyDBLite。

Tornado

一个轻量级的Web框架，内置非阻塞式服务器，而且速度相当快

webpy

一个小巧灵活的Web框架，虽然简单但是功能强大。

CherryPy

基于Python的Web应用程序开发框架。

Pylons

基于Python的一个极其高效和可靠的Web开发框架。

Zope

开源的Web应用服务器。

TurboGears

基于Python的MVC风格的Web应用程序框架。

Twisted

流行的网络编程库，大型Web框架。

Quixote

Web开发框架。

aiohttp

轻量级的Web框架，采用的是Python3的asyncio异步特性。

Pandas

非常高效的数据科学库

* 科学计算

Matplotlib

用Python实现的类matlab的第三方库，用以绘制一些高质量的数学二维图形。

Pandas

用于数据分析、数据建模、数据可视化的第三方库。

SciPy

基于Python的matlab实现，旨在实现matlab的所有功能。

NumPy

基于Python的科学计算第三方库，提供了矩阵，线性代数，傅立叶变换等等的解决方案。

GUI[编辑]

PyGtk

基于Python的GUI程序开发GTK+库。

PyQt

用于Python的QT开发库。

WxPython

Python下的GUI编程框架，与MFC的架构相似。

* 其它

BeautifulSoup

基于Python的HTML/XML解析器，简单易用。

gevent

python的一个高性能并发框架,使用了epoll事件监听、协程等机制将异步调用封装为同步调用。

PIL

基于Python的图像处理库，功能强大，对图形文件的格式支持广泛。目前已无维护，另一个第三方库Pillow实现了对PIL库的支持和维护。

PyGame

基于Python的多媒体开发和游戏软件开发模块。

Py2exe

将python脚本转换为windows上可以独立运行的可执行程序。

Requests

适合于人类使用的HTTP库，封装了许多繁琐的HTTP功能，极大地简化了HTTP请求所需要的代码量。

scikit-learn

机器学习第三方库，实现许多知名的机器学习演算法。

TensorFlow

Google开发维护的开源机器学习库。

Keras

基于TensorFlow，Theano与CNTK的高阶神经网路API。

SQLAlchemy

关系型数据库的对象关系映射(ORM)工具

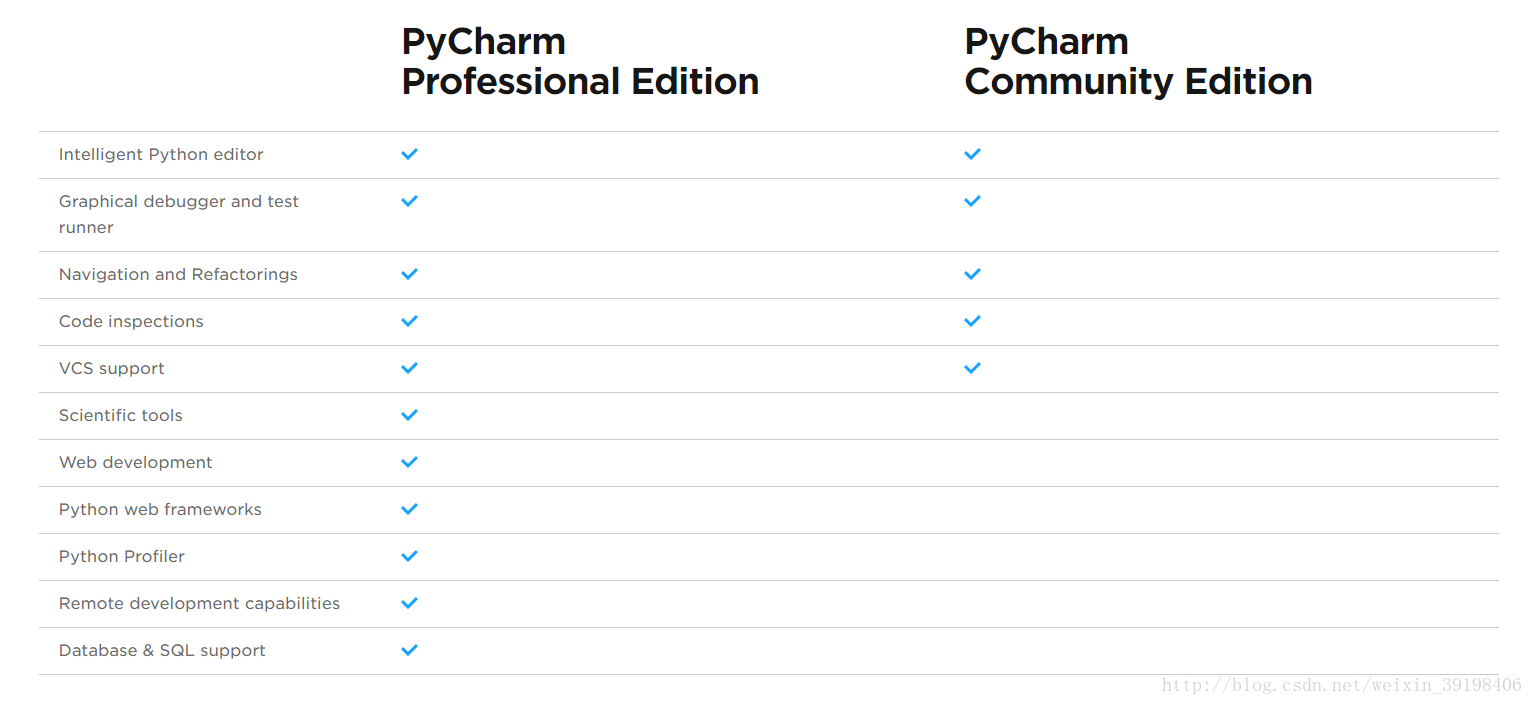
## 第二章Python环境搭建 pycharm安装

1. Python安装

环境变量配置 (包含python.exe和 pip.exe路径)

这时就可以使用Python执行py文件了

1. pycharm安装



1. 创建项目
2. 创建虚拟环境
3. 打开项目
4. 打印hello world

## 第三章Python基础语法

### 3.1 Python3的基础语法

#### 编码

默认情况下，Python 3 源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。 当然你也可以为源码文件指定不同的编码：

*# -\*-coding:utf-8 -\*-*

#### 标识符

1、第一个字符必须是字母表中字母或下划线'\_'。

2、标识符的其他的部分有字母、数字和下划线组成。

4、标识符对大小写敏感。

其中，一般情况下，类名为单词首字母大写的驼峰式命名，全局变量和常量为所有字母大写。其他的，如包名、模块名、文件名、变量名、函数名使用小写字母配合下划线使用

#### 保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python 的标准库提供了一个 keyword 模块，可以输出当前版本的所有关键字：

*# -\*-coding:utf-8 -\*-***import** keyword  
print(keyword.kwlist)

运行结果：

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

#### 注释

1、Python中单行注释以 # 开头，实例如下：

*# -\*- coding:utf-8 -\*-  
#第一个python程序*print(**"Hello,Python"**) *#打印结果*

2、批量、多行注释符号

在python中也会有注释有很多行的时候，这种情况下就需要批量多行注释符了。多行注释是用三引号'''   '''或””” “””包含的，例如：

**'''  
该方法调用go方法  
需要返回一个字符串类型  
'''**

**或**

**"""  
该方法调用go方法  
需要返回一个字符串类型  
"""**

#### 行与缩进

python最具特色的就是使用缩进来表示代码块，不需要使用大括号({})。

缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

pep8 规范要求Python缩进为4个空格

**if True** :  
 print(**"is True"**)  
**else** :  
 print(**"is False"**)

如果出现缩进不一致的情况会出现，直接报错

**if True** :  
 print(**"is True"**)  
**else** :  
 print(**"is False"**)  
 print(**"Error"**)

File "D:/Just-Do-IT/WorkSpace/PythonSpace/test/first.py", line 15

print("00000000")

^

IndentationError: unexpected indent

#### 多行语句

Python 通常是一行写完一条语句，但如果语句很长，我们可以使用反斜杠(\)来实现多行语句（不推荐），例如：

a = 10  
b = 20  
c = 30  
  
sum = a + \  
 + b \  
 + c  
print(sum)

在 [], {}, 或 () 中的多行语句，不需要使用反斜杠(\)，例如：

sum = [a,  
 b , c]  
print(sum)

#### 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是Python语法的一部分。书写时不插入空行，Python解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

#### 等待用户输入input

line = input(**"请输入,按下enter键后退出:"**)  
print(line)  
print(type(line))

以上代码中 ，打印"请输入,按下enter键后退出:"。一旦用户按下键时，程序将退出。

**注意：**在以前的2.x版本中有raw\_input()这个函数，而在现在的3.x版本中，因为版本进行了升级所以raw\_put()已经被废弃掉了。这两个函数在2.x的版本中是有区别的。

### 3.2 数据类型与运算符

#### 3.2.1 基本数据类型

Python 中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

在 Python 中，变量就是变量，它没有类型，我们所说的"类型"是变量所指的内存中对象的类型。

等号（=）用来给变量赋值。

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

*# -\*- coding: utf-8 -\*-*count = 1 *#整型变量*flag = **True** *#布尔型*name = **" FBQ "** *#字符串类型*fnum = 12.5 *#浮点型*print(type(count))  
print(type(flag))  
print(type(name))  
print(type(fnum))

运行结果：

<class 'int'>

<class 'bool'>

<class 'str'>

<class 'float'>

##### 如何为多个变量赋值？

同时为多个变量赋一个值

a = b = c = 1

您也可以为多个对象指定多个变量。

a , b , c = 1 , 2 , **"Python"**print(a,b,c)

运行结果：

1 2 Python

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

例：Python基本数据类型所占用的内存空间的大小

*# -\*- coding: utf-8 -\*-***import** sys  
  
count = 1 *#整型变量*flag = **True** *#布尔型*name = **"FBQ"** *#字符串类型*fnum = 12.5 *#浮点型*print(**"%s Size %d"** %(type(count),sys.getsizeof(count)))  
print(**"%s Size %d"** %(type(flag),sys.getsizeof(flag)))  
print(**"%s Size %d"** %(type(name),sys.getsizeof(name)))  
print(**"%s Size %d"** %(type(fnum),sys.getsizeof(fnum)))

#### 3.2.2 标准数据类型

Python3 中有六个标准的数据类型：

1、Number（数字）

2、String（字符串）

3、List（列表）

4、Tuple（元组）

5、Sets（集合）

6、Dictionary（字典）

##### 1） Number（数字）

Python3 支持 int、float、bool、complex（复数）。

在Python 3里，只有一种整数类型 int，表示为长整型，没有 python2 中的 Long。像大多数语言一样，数值类型的赋值和计算都是很直观的。内置的 type() 函数可以用来查询变量所指的对象类型。

a , b ,c , d = 20 , 5.5 , **True** , 4 + 3j  
print(type(a),type(b),type(c),type(d))

<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>

此外还可以用 isinstance 来判断：

a = 1  
data\_type1 = isinstance(a,int)  
data\_type2 = isinstance(a,bool)  
print(data\_type1,data\_type2)

运行结果：

True False

isinstance 和 type 的区别在于：

*# 以继承的形式定义两个类*

**class** A(object):  
 **pass  
class** B(A):  
 **pass**print(isinstance(A(),A),type(A())==A)  
print(isinstance(B(),A),type(B())==A)

运行结果：

True True

True False

区别就是:

type()不会认为子类是一种父类类型。

isinstance()会认为子类是一种父类类型。

**注意：**在 Python2 中是没有布尔型的，它用数字 0 表示 False，用 1 表示 True。到 Python3 中，把 True 和 False 定义成关键字了，但它们的值还是 1 和 0，它们可以和数字相加。

当你指定一个值时，Number 对象就会被创建：

var1 = 20;  
var2 = 21;

还可以使用del语句删除一些对象引用。

del语句的语法是：

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

var1 = 20;  
var2 = 21;  
  
**del** var1 , var2

**数值运算实例：**

print(5 + 4) *#加法*print(4.3 - 2) *#减法*print(3 \* 7) *#乘法*print(2 / 4) *#除法，得到一个浮点数*print(2 // 4) *#除法，得到整数*print(17 % 3) *#取余*print(2 \*\* 6) *#乘方*

小结：

1、Python可以同时为多个变量赋值，如a, b = 1, 2。

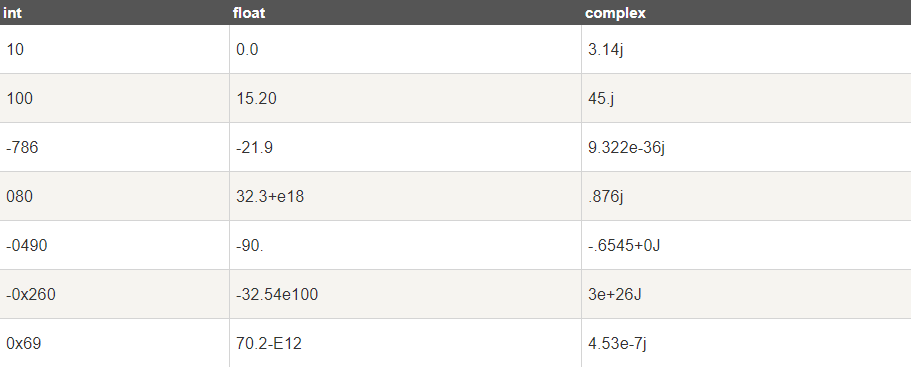
2、一个变量可以通过赋值指向不同类型的对象。

3、数值的除法（/）总是返回一个浮点数，要获取整数使用//操作符。

4、在混合计算时，Python会把整型转换成为浮点数。

5、Python中除了''、""、0、()、[]、{}、None为False之外，其他的都是True。

**数值类型实例**



Python还支持复数，复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型。

##### 2） String（字符串）

Python中的字符串用单引号(')或双引号(")括起来

索引值以 0 为开始值，-1 为从末尾的开始位置。

字符串的截取的语法格式如下：

变量[头下标:尾下标]

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

**例:**

str = **"python"***#打印字符串*print(str)  
*#输出第一个到倒数第二个的所有字符*print(str[0:-1])  
*# 输出字符串第一个字符*print(str[0])  
*# 输出从第1个开始到第3个的字符*print(str[0:4])  
*#输出从第1个往后的全部字符串*print(str[1:])  
*#输出当前字符串两次*print((str+**" "**) \* 2)  
others = **"Tensorflow"**print(str + others)

运行结果：

python

pytho

p

pyth

ython

python python

pythonTensorflow

python使用反斜杠(\)转义特殊字符。

例：

*#加上\转义特殊字符*

str = **"\n"**print(**"------------"**)  
print(str)  
print(**"------------"**)

除了换行符\n，常用的还有：反斜杠符号\\，单引号\’（针对单引号字符串），双引号\” （针对双引号字符串），制表符\t，回车\r

如果想使用转义形式的非转义字符，则只需在字符串前面加一个r/R即可

例：

*#加上r/R表示原始字符串*

str = **r'\n'**print(str)

另外，反斜杠(\)可以作为续行符，表示下一行是上一行的延续。也可以使用 """...""" 或者 '''...''' 跨越多行。

str = **'''py  
thon'''**print(str)  
  
str = **'p\  
 ython'**print(str)

注意：Python 没有单独的字符类型，一个字符就是长度为1的字符串。

与 C 字符串不同的是，Python 字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。

如下：

word = **"python"**word[0] = **"a"**

**如何迭代一个字符串？**

words = **"python"  
for** word **in** words:  
 print(word)

**如何判断一个字符串是否包含(不包含)另一个字符串?**

python提供了in和not in关键字

a = **"python"**b = **"Hello python"**print(a **in** b)  
print(a **not in** b)

运行结果：

True

False

小结：

1. 反斜杠可以用来转义，使用r可以让反斜杠不发生转义。
2. 字符串可以用+运算符连接在一起，用\*运算符重复。
3. Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。
4. Python中的字符串不能改变。

**字符串格式化**

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

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

**例:**

print(**"人生苦短，我用%s"**%(**"python。"**))  
print(**"1 + 1 = %s 或 %d"**%(**"田"**,2))

运行结果：

人生苦短，我用python。

1 + 1 = 田 或 2

**python字符串格式化符号:**



**Unicode 字符串**

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

在Python3中，所有的字符串都是Unicode字符串。

**字符串内建函数**

1. capitalize()：将字符串的第一个字符转换为大写。

s = **"python"**  
print(s.capitalize())

1. center(width, fillchar)：返回一个指定的宽度 width 居中的字符串，fillchar 为填充的字符，默认为空格。

s = **"python"**print(s.center(20,**"\*"**))

1. 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

s = **"python"**print(s.count(**"t"**,1,len(s)))

1. 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。errors -- 设置不同错误的处理方案。默认为 'strict',意为编码错误引起一个UnicodeError。 其他可能得值有 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' 以及通过 codecs.register\_error() 注册的任何值。

s = **"人生苦短，我用python。"**s\_utf = s.encode(**"UTF-8"**)  
s\_gbk = s.encode(**"GBK"**)

*#输出16进制的bytes对象*

print(s\_utf)  
print(s\_gbk)

print(s\_utf.decode(**"utf-8"**))  
print(s\_gbk.decode(**"gbk"**))

运行结果：

b'\xe4\xba\xba\xe7\x94\x9f\xe8\x8b\xa6\xe7\x9f\xad\xef\xbc\x8c\xe6\x88\x91\xe7\x94\xa8python\xe3\x80\x82'

b'\xc8\xcb\xc9\xfa\xbf\xe0\xb6\xcc\xa3\xac\xce\xd2\xd3\xc3python\xa1\xa3'

1. endswith(suffix, beg=0, end=len(string))：检查字符串是否以 suffix结束，如果beg 或者 end 指定则检查指定的范围内是否以 suffix结束，如果是，返回 True,否则返回 False。

s = **"人生苦短，我用python。"**print(s.endswith(**"python。"** , 0 , len(s)))

1. expandtabs(tabsize=8)：把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8 。tabsize -- 指定转换字符串中的 tab 符号('\t')转为空格的字符数。

s = **"人生苦\t短，我\t用python。"**print(s)  
print(s.expandtabs(1))

1. find(str, beg=0 end=len(string))：检测 str 是否包含在字符串中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1。

s = **"人生苦短，我用python。"**print(s.find(**"苦"** , 1 , len(s)))

1. 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。

s = **"人生苦短，我用python。"**print(s.index(**"苦"** , 0 , len(s)))

没有字符串抛出异常：

ValueError: substring not found

1. 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。(方法检测字符串是否由字母和数字组成)

s1 = **"2019python"**print(s1.isalnum())  
s2 = **"python."**print(s2.isalnum())

1. 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。

s1 = **"python"**print(s1.isalpha())  
s2 = **"python2019"**print(s2.isalpha())

1. isdigit()：如果字符串**只**包含数字则返回 True 否则返回 False。

s = **"2019"**print(s.isdigit())

1. 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。

s = **"人生苦短，我用python。"**print(s.islower())

1. isnumeric()：如果字符串中只包含数字字符，则返回 True，否则返回 False。

s = **"2019"**print(s.isnumeric())

1. isspace()：如果字符串中只包含空格，则返回 True，否则返回 False。

s = **" "**print(s.isspace())

1. istitle()：如果字符串是标题化的(见 title())则返回 True，否则返回 False。(istitle() 方法检测字符串中所有的单词拼写首字母是否为大写，且其他字母为小写。)

s = **"Life Is Short, I Use Python."**print(s.istitle())

1. 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。

s = **"人生苦短，我用PYTHON。"**print(s.isupper())

1. 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。

s = **"-"**seq = (**"p"**,**"y"**,**"t"**,**"h"**,**"o"**,**"n"**)  
print(s.join(seq))

1. len(string)：判断字符串的长度。

s = **"人生苦短，我用PYTHON。"**print(len(s))

1. lower()\upper()：转换字符串中所有大\小写字符为小\大写。

s = **"人生苦短，我用PYTHON。"**print(s.lower())

s = **"Life Is Short, I Use Python."**print(s.upper())

1. lstrip()、rstrip()、strip([chars])：截掉字符串左边的空格。

strip() 方法用于移除字符串头尾指定的字符（默认为空格或换行符）或字符序列。

注意：该方法只能删除开头或是结尾的字符，不能删除中间部分的字符。

s = **" 人生苦短，我用PYTHON。 ...."**print(s.lstrip())

s = **"pyt hon "**print(s.rstrip())

*#移除p、n和空格*

s = **"pyt hon "**print(s.strip(**"pn "**))

1. 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。

intab = **"abc"**outtab = **"123"**s1 = str.maketrans(intab,outtab)  
str = **"this is string example....wow!!!"**print(str.translate(s1))

1. max(str)\min(str)：返回字符串 str 中最大\小的字母。

s = **"abcdr"**print(max(s))  
print(min(s))

1. 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。

s = **"abcadar"**print(s.replace(**"a"**,**"b"**,2))

1. rfind(str, beg=0,end=len(string))：类似于 find()函数，不过是从右边开始查找.

s = **"abcadar"**print(s.rfind(**"a"**,0,-1))

1. rindex( str, beg=0, end=len(string))：类似于 index()，不过是从右边开始.

s = **"abcadar"**print(s.rindex(**"a"**,0,-1))

没有找到会报错

1. 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

s = **"python"**print(s.rjust(10,**"\*"**))

s = **"python"**print(s.ljust(10,**"\*"**))

1. 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 个子字符串。

s = **"p y t h o n"**print(s.split(**" "**))  
print(s.split(**" "**,3))

s = **"python\ndu"**print(s.split(**"\n"**))

1. 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。

s = **"ab c\n\nde fg\rkl\r\n"**print(s.splitlines())  
print(s.splitlines(**True**))

1. startswith(str, beg=0,end=len(string))：检查字符串是否是以 obj 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。

s = **"人生苦短，我用python。"**print(s.startswith(**"人"**,0,-1))

1. swapcase():将字符串中大写转换为小写，小写转换为大写。

s = **"PYTHON人生苦短，我用python。"**print(s.swapcase())

1. 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。

table -- 翻译表，翻译表是通过maketrans方法转换而来。

deletechars -- 字符串中要过滤的字符列表。

s1 = **"abcdefg"**s2 = **"1234567"**m = str.maketrans(s1,s2)  
print(m)  
s3 = **"abcder13"**.translate(m)  
print(s3)

1. zfill (width)：返回长度为 width 的字符串，原字符串右对齐，前面填充0。

s = **"python"**print(s.zfill(10))

1. 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。

s1 = **"abcd"**s2 = **"2017"**print(s1.isdecimal())  
print(s2.isdecimal())

##### 3） List（列表）

List（列表） 是 Python 中使用最频繁的数据类型之一。

列表可以完成大多数集合类的数据结构实现。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（所谓嵌套）。

列表是写在方括号([])之间、用逗号分隔开的元素列表。

和字符串一样，列表同样可以被索引和截取，列表被截取后返回一个包含所需元素的新列表。

列表截取的语法格式如下：

变量[头下标:尾下标]

Python有6个序列的内置类型即列表、元组、字符串、Unicode字符串、buffer对象对象，但最常见的是列表和元组。

1. 创建一个列表

l1 = [1,2,3,4,5]  
l2 = [**"python"**,**"java"**,**"google"**]  
print(l1,l2)

*#截取生成一个新列表*

print(l1[0:1])

与字符串的索引一样，列表索引从0开始。列表可以进行截取、组合等。

1. 访问列表中的值

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

l1 = [1,2,3,4,5]  
l2 = [**"python"**,**"java"**,**"google"**]  
*#获取l1中的第三个元素*print(l1[2])  
*#获取l2中第2个到最后一个元素*print(l2[1:])

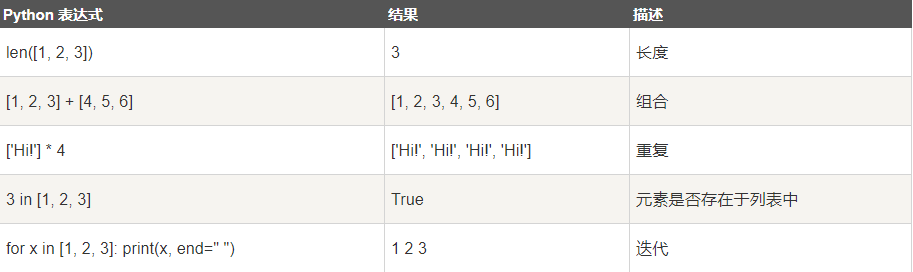
1. 更新列表：

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

l1 = [1,2,3,4,5]  
l2 = [**"python"**,**"java"**,**"google"**]  
l2[1] = **"C"**print(l2)  
*#一次批量更新*l1[0:3] = [0,0,0]  
print(l1)

**Python列表脚本操作符：**

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



**嵌套列表：**

l = [[1,2,3,4,5],[**"A"**,**"B"**,**"C"**]]  
print(l)  
print(l[0][2:4])  
print(l[1][2])

**Python列表函数：**

1. len()： (获取元素个数)

l = [1,2,3,4,5]  
print(len(l))  
**for** x **in** l : print(x)

1. max(list)\min(list)：(获取最大/最小值，字符串根据首字母的ASCII码进行比较)

l = [1,2,3,4,5]  
print(max(l))  
l1 = [**"A"**,**"b"**,**"C"**]  
print(max(l1))  
print(min(l1))

1. list(seq)：将元组和字符串转换为列表。

t = (**"python"**,**"java"**,(**"C"**,**"C++"**,**"Ruby"**))  
l = list(**"python"**)  
print(list(t))  
print(l)

1. list.append(obj)：在列表末尾添加新的对象。

l = [**"python"**,**"google"**,**"c"**]  
l.append(100)  
l.append(**"你好"**)  
print(l)

1. list.count(obj)：统计某个元素在列表中出现的次数。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
print(l.count(**"python"**))

1. 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）。

l = [**"python"**,**"google"**,**"c"**,**"python"**]

*#把字符串转化为列表，追加*  
print(l.extend(**"c++"**))  
print(l)

*#用列表生成器生成列表，追加*  
l1 = list(range(5))  
l.extend(l1)  
print(l)

1. list.index(obj)：获取对象的索引。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
index = l.index(**"python"**)  
print(index)

1. list.insert(index, obj)：将对象插入列表。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.insert(1,**"java"**)  
print(l)

9、 list.pop(obj=list[-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。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.pop()  
print(l)  
l.pop(1)  
print(l)

1. list.remove(obj)：移除列表中某个值的第一个匹配项。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.remove(**"python"**)  
print(l)

1. list.reverse()：反向列表中元素。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.reverse()  
print(l)

1. list.sort([func])：对原列表进行排序，func为可选函数。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.sort()  
print(l)

1. list.clear()：清空列表。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
print(l)  
l.clear()  
print(l)

1. list.copy()：复制列表 ，如果列表中嵌套一个列表，对子列表进行更改，则新旧列表的区别？

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l1 = l.copy()  
print(l1)  
l1.append(**"java"**)  
print(**"原："**+**""**.join(l))  
print(**"新："**+**""**.join(l1))

##### 4） Tuple(元组)

Python 的元组与列表类似，不同之处在于元组的元素不能修改。

元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

1. 创建元组

t = (**"python"**,**"google"**,**"c"**,**"python"**)  
print(t)

1. 创建一个空元组

t = ()  
print(t)

1. 创建单元素元组

元组中只包含一个元素时，需要在元素后面添加逗号，否则该元组会被识别为字符串。

t = (**"python"**,)  
print(t)

元组与字符串类似，下标索引从0开始，可以进行截取，组合等。

1. 访问元组

t1 = (1,2,3,4,5,6,)  
t2 = (**"python"**,**"google"**,**"java"**,**"A"**)  
print(t1[1])  
print(t2[2])  
print(t1[3:60])

**元组中的元素值是不允许修改的，但我们可以对元组进行连接组合。**

t1 = (1,2,3,4,5,6,)  
t2 = (**"python"**,**"google"**,**"java"**,**"A"**)  
print(t1 + t2)

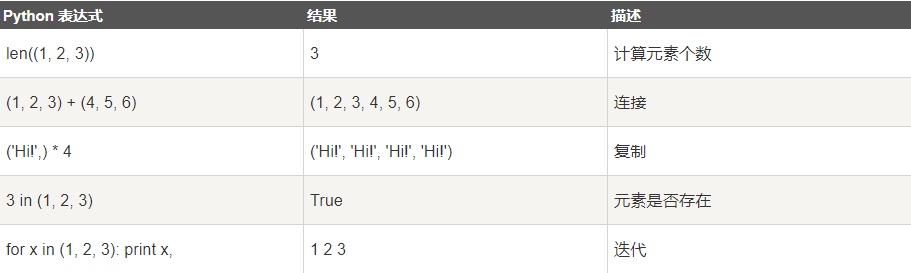
1. 删除元组：

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组。

t1 = (1,2,3,4,5,6,)  
print(t1)  
**del** t1  
print(t1)

**元组运算符：**

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



**元组内置函数：**

1. len(tuple)：计算元组的长度。
2. max(tuple)：查找元组中最大的元素。
3. min(tuple)：查找元组中最小的元素。
4. tuple(seq)：将列表转换为元组。

##### 5） Dict(字典)

字典是另一种可变容器模型，且可存储任意类型对象。

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

1. 创建字典

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

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(d1)  
d2 = {1:**"A"**,**"B"**:2,(1,2,3):**"c"**}  
print(d2)

1. 访问字典里的值

print(d1[**"Java"**])  
d2 = {1:**"A"**,**"B"**:2,(1,2,3):**"c"**}  
print(d2[(1,2,3)])

1. 修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
d1[**"C++"**] = 7000  
d1[**"Ruby"**] = 3000  
print(**"修改添加数据："**,d1)  
**del** d1[**"Java"**]  
print(**"删除数据："**,d1)  
**del** d1  
d1

**字典键的特性：**

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

需要注意以下两点：

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，只有后一个值会被记住。

2）键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行。

**字典内置函数：**

1. len(dict)：计算字典元素个数，即有效键值对的总数（重复的键只能算一个）。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(len(d1))

1. str(dict)：输出字典，以可打印的字符串表示。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(str(d1))

print(d1.\_\_str\_\_())

print(json.dumps(d2)) *#该方法需要保证key都为字符串或数字*

1. type(variable)：返回输入的变量类型，如果变量是字典就返回字典类型。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(type(d1))

1. dict.clear()：删除字典内所有元素。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(d1)  
d1.clear()  
print(d1)

1. dict.copy()：返回字典的浅复制。

直接赋值：其实就是对象的引用（别名）。

浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。

深拷贝(deepcopy)： copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。

**浅copy()**

a = {1:[1,2,3]}  
b = a.copy()  
a[1].remove(1)  
print(a,b)

**深copy()**

**import** copy  
  
a = {1:[1,2,3]}  
b = copy.deepcopy(a)  
a[1].append(4)  
print(a,b)

**解析：**

b = a: 赋值引用，a 和 b 都指向同一个对象。

b = a.copy(): 浅拷贝, a 和 b 是一个独立的对象，但他们的子对象还是指向统一对象（是引用）。

b = copy.deepcopy(a): 深度拷贝, a 和 b 完全拷贝了父对象及其子对象，两者是完全独立的。

1. dict.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。

d = dict.fromkeys((1,2,3,4),[1,2,3,4])  
print(d)

1. dict.get(key, default=None)：返回指定键的值，如果值不在字典中返回default值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.get(1,**"没有"**))

1. key in dict：如果键在字典dict里返回true，否则返回false。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(4 **in** d1)

1. dict.items()：以列表返回可遍历的(键, 值) 元组数组。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
**for** item **in** d1.items():  
 print(item)

10、dict.keys()：以列表返回一个字典所有的键。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
keys = d1.keys()  
**for** key **in** keys:  
 print(d1.get(key))

11、.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。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
d1.setdefault(4,**"D"**)  
print(d1)

1. dict.update(dict2)：把字典dict2的键/值对更新到dict里。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
d2 = {4:**"D"**}  
d3 = {1:**"B"**,2:**"A"**}  
d1.update(d2)  
print(d1)  
d1.update(d3)  
print(d1)

1. dict.values()：以列表返回字典中的所有值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.values())

1. 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。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.pop(1))  
print(d1)

1. popitem()：随机返回并删除字典中的一对键和值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.popitem())  
print(d1)  
print(d1.popitem())

#### 3.2.3 运算符

**什么是运算符？**

1+3 = 4 其中的1和3是操作数，其中的 “+”就是运算符。

Python语言支持以下类型的运算符:

1、算术运算符

2、比较（关系）运算符

3、赋值运算符

4、逻辑运算符

5、位运算符

6、成员运算符

7、身份运算符

8、运算符优先级

**Python算术运算符：**



**Python比较运算符：**



**Python赋值运算符：**



**Python位运算符：**



变量 a 为 60，b 为 13二进制格式

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

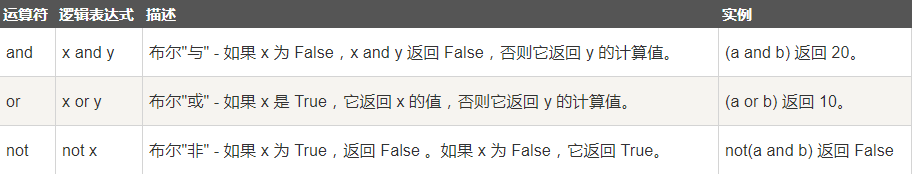
a^b = 0011 0001

~a = 1100 0011

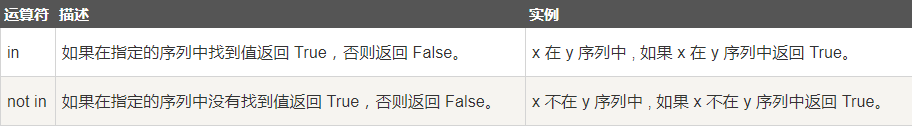
例：

a = 60 *# 60 = 0011 1100*b = 13 *# 13 = 0000 1101*c = 0  
  
c = a & b; *# 12 = 0000 1100*print(**"1 - c 的值为："**, c)  
  
c = a | b; *# 61 = 0011 1101*print(**"2 - c 的值为："**, c)  
  
c = a ^ b; *# 49 = 0011 0001*print(**"3 - c 的值为："**, c)  
  
c = ~a; *# -61 = 1100 0011*print(**"4 - c 的值为："**, c)  
  
c = a << 2; *# 240 = 1111 0000*print(**"5 - c 的值为："**, c)  
  
c = a >> 2; *# 15 = 0000 1111*print(**"6 - c 的值为："**, c)

**Python逻辑运算符：**

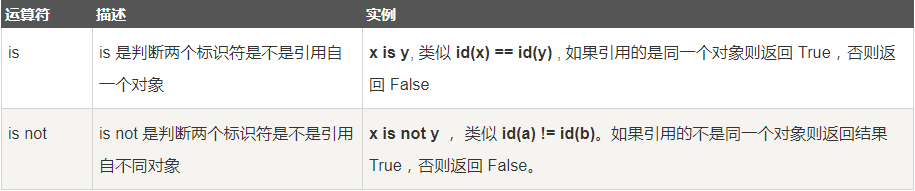


**Python成员运算符：**



**Python身份运算符：**

用于比较两个对象的存储单元



id() 函数用于获取对象内存地址。

is 与 == 区别：

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

**Python运算符优先级：**

从最高到最低优先级的所有运算符：



### 3.3函数

#### 3.3.1什么是函数？

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

#### 3.3.2 如何定义一个函数？

你可以定义一个由自己想要功能的函数，以下是简单的规则：

1、函数代码块以 def 关键词开头，后接函数标识符名称和圆括号()。

2、任何传入参数和自变量必须放在圆括号中间。

3、函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。

4、函数内容以冒号起始，并且缩进。

5、return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

*#-\*-coding:utf-8-\*-***def** func():  
 print(**"定义第一个函数！！！！"**)

定义一个有返回值的函数

**def** func():  
 print(**"定义第一个函数！！！！"**)  
 **return "我是函数"**,20  
  
par1,par2 = func()  
print(par1)  
print(par2)

#### 3.3.3参数传递

*#-\*-coding:utf-8-\*-*a = [1,2,3]  
a = **"tensorflow"**

[1,2,3]是List类型而**"tensorflow"**是String类型，这里的a是没有类型的，他仅仅是一个对象的引用(一个指针)，他既可以指向List类型的对象，也可以指向String类型的对象。

**可更改(mutable)与不可更改(immutable)对象**

在 python中，String, tuple,和number是不可更改的对象，而 list,dict 等则是可以修改的对象。

这里会具体解释：

**不可变类型**：变量赋值a=5后再赋值a=10，这里实际是新生成一个int值对象10，再让a指向它，而5被丢弃，不是改变a的值，相当于新生成了a。

**可变类型**：变量赋值 l=[1,2,3,4] 后再赋值 l[2]=5 则是将 list l的第三个元素值更改，本身l没有动，只是其内部的一部分值被修改了。

**python 函数的参数传递：**

**不可变类型：**类似 c++ 的值传递，如整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。

**可变类型：**类似 c++ 的引用传递，如 列表，字典。如 fun（l），则是将l真正的传过去，修改后fun外部的l也会受影响。

python中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

**python 传不可变对象实例**

*#-\*-coding:utf-8-\*-***def** change\_int(b):  
 b = 2  
b = 1  
change\_int(b)  
print(b)

实例中有 int 对象 1，指向它的变量是 b，在传递给 change\_int函数时，按传值的方式复制了变量 b，change\_int中的b 和 b 都指向了同一个 Int 对象，在change\_int中的b =2 时，则新生成一个 int 值对象 2，并让change\_int中的b 指向它。

**Python传可变对象实例**

*#-\*-coding:utf-8-\*-***def** no\_change(l):  
 l.append(**"A"**)  
 print(l)  
  
l = [1,2,3]  
no\_change(l)  
print(l)

**不定长参数**

有时可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，声明时不会命名。

**def** print\_demo(args,\*data):  
 print(args)  
 **for** word **in** data:  
 print(word)  
  
printDemo(**"hello"**,1,2,3,4)

注：参数以一个＊号开头的代表着一个任意长度的元组(tuple)，可以接收连续一串参数；  
参数以两个＊号开头的代表着一个字典，参数的形式是“key=value”，接受连续任意多个参数。

**def** print\_demo(arg,\*args,\*\*dic):  
 print(arg)  
 print(**"------------"**)  
 **for** item **in** args:  
 print(item)  
 print(**"------------"**)  
 print(dic)  
 print(dic[**"a"**])  
  
  
print\_demo(**"args"**, **"a"**, **"b"**, **"c"**, 4, 5, 6, a=**"a"**, c=**"b"**, d=**"d"**)

### 3.4 条件控制与循环 迭代器与生成器

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

#### 3.4.1 if 语句

Python中if语句的一般形式如下所示：

if condition\_1: statement\_block\_1

elif condition\_2: statement\_block\_2

else: statement\_block\_3

如果 "condition\_1" 为 True 将执行 "statement\_block\_1" 块语句

如果 "condition\_1" 为False，将判断 "condition\_2"

如果"condition\_2" 为 True 将执行 "statement\_block\_2" 块语句

如果 "condition\_2" 为False，将执行"statement\_block\_3"块语句

Python 中用 elif 代替了 else if，所以if语句的关键字为：if – elif – else。

**注意：**

1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。

2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。

3、在Python中没有switch – case语句。

例1：

a = 100  
b = 0  
**if** a:  
 print(**"a条件为True"**)  
  
**if** b:  
 print(**"b条件为True"**)

这时候呢if b不执行

例2：

判断年龄  
age = int(input(**"请输入年龄:"**))  
**if** age <18:  
 print(**"未成年人"**)  
**elif** age <40:  
 print(**"年轻人"**)  
**elif** age < 60:  
 print(**"中年人"**)  
**else**:  
 print(**"老年人"**)

if 嵌套：

if 表达式1:

语句

if 表达式2:

语句

elif 表达式3:

语句

else:

语句

elif 表达式4:

语句

else:

语句

#### 3.4.2 循环语句

Python中的循环语句有 for 和 while。

Python循环语句的控制结构图如下所示：

while 循环：

while 判断条件：

语句

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

以下实例使用了 while 来计算 1 到 100 的总和：

count = 100  
sum = 0;  
**while** count:  
 sum += count  
 count -= 1  
  
  
print(sum)

无限循环

我们可以通过设置条件表达式永远不为 false 来实现无限循环。

**while** 1 == 1:  
 s = input(**"请输入内容："**)  
 print(s)  
 **if** s == **"exit"**:  
 **break**print(**"Good Bye"**)

while 循环使用 else 语句：

在 while … else 在条件语句为false时或者循环结束时，执行 else 的语句块：

count = 1  
**while** count < 5:  
 print(str(count)+**"小于5"**)  
 count += 1  
**else**:  
 print(str(count) +**"大于等于5"**)

简单语句组：

类似if语句的语法，如果你的while循环体中只有一条语句，你可以将该语句与while写在同一行中。

flag = **True  
  
while** flag: print(**"人生苦短，我用python！"**)  
print(**"Good Bye"**)

for 语句

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

for循环的一般格式如下：

for <variable> in <sequence>:

<statements>

else: <statements>

items = [1,2,3,4,5,6]  
**for** item **in** items:  
 print(item)

range()函数

**for** i **in** range(10):  
 print(i)  
  
print(**"==============================="**)  
**for** j **in** range(5,10):  
 print(j)

可以使range以指定数字开始并指定不同的增量(甚至可以是负数，有时这也叫做'步长'):

**for** i **in** range(1,20,2):  
 print(i)

**for** i **in** range(20,1,-1):  
 print(i)

break和continue语句及循环中的else子句：

break 语句可以跳出 for 和 while 的循环体。如果你从 for 或 while 循环中终止，任何对应的循环, else 块将不执行。

**for** i **in** range(20,1,-1):  
 **if** i == 6:  
 **break** print(i)  
print(**"Break"**)

continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

**for** i **in** range(20,1,-1):  
 **if** i == 6:  
 **continue** print(i)  
print(**"continue"**)

pass 语句：

Python pass是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句。

例：

**if True**:  
 **pass  
  
for** i **in** range(6):  
 **pass  
  
while True**:  
 **pass  
  
class** Pass(object):  
 **pass  
  
  
def** go():  
 **pass**

#### 3.4.3迭代器

**什么是迭代？**

可以直接作用于for循环的对象统称为可迭代对象(Iterable)。

可以被next()函数调用并不断返回下一个值的对象称为迭代器(Iterator)。

所有的Iterable均可以通过内置函数iter()来转变为Iterator。

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

字符串，列表或元组对象都可用于创建迭代器：

l = [1,2,3,4,5,6,7]  
it = iter(l)  
  
print(next(it))  
print(next(it))  
print(next(it))  
print(next(it))

迭代器对象可以使用常规for语句进行遍历：

l = [1,2,3,4,5,6,7]  
it = iter(l)  
  
**for** i **in** it:  
 print(i,end=**"\n"**)

使用next()函数：

**import** sys  
  
l = [1,2,3,4,5,6,7]  
it = iter(l)  
  
**while True**:  
 **try**:  
 print(next(it))  
 **except** StopIteration: *#迭代器没有更多的值* sys.exit()

题外话：

1. 内置函数iter()仅仅是调用了对象的\_\_iter()方法，所以list对象内部一定存在方法iter\_\_()
2. 内置函数next()仅仅是调用了对象的\_\_next()方法，所以list对象内部一定不存在方法next\_\_()，但是Itrator中一定存在这个方法。
3. for循环内部事实上就是先调用iter()把Iterable变成Iterator再进行循环迭代的。

#### 3.4.4 生成器

列表生成式：

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

例：我们要生成要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

l = list(range(1,11))  
print(l)

例：生成[1, 4, 9,16 ..., 100]怎么做？

方法一：

l = []  
**for** i **in** range(1,11):  
 l.append(i \* i)  
  
print(l)

方法二：

我们来用列表生成式来做：

l = [i \* i **for** i **in** range(1,11)]  
print(l)

例：还可以使用两层循环，可以生成全排列：

l = [a + b **for** a **in "ABC" for** b **in "ZYX"**]  
print(l)

例：列出当前目录下的所有文件和目录名：

**import** os  
l = [file **for** file **in** os.listdir(**"C:/"**)]  
print(l)

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

l = [i \* i **for** i **in** range(1,11)]  
print(type(l))  
  
print(l)  
l = (i \* i **for** i **in** range(1,11))  
print(type(l))  
print(l)  
print(next(l))

generator保存的是算法，每次调用next(l)，就计算出l的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(l)实在是（变态），正确的方法是使用for循环，因为generator也是可迭代对象。

所以，我们创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波那契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波那契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

**def** fib(num):  
 n , a , b = 0 , 0 , 1  
 **while** n < num :  
 print(b,end=**" "**)  
 a , b = b , a + b  
 n = n + 1  
  
fib(6)

仔细观察，可以看出，fib函数实际上是定义了斐波那契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

**def** fib(num):  
 n , a , b = 0 , 0 , 1  
 **while** n < num :  
 **yield** b  
 a , b = b , a + b  
 n = n + 1  
  
**for** i **in** fib(6):  
 print(i)

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

**def** pop():  
 print(**"step1"**)  
 **yield** 1  
 print(**"step2"**)  
 **yield** 3  
 print(**"step3"**)  
 **yield** 5  
  
p = pop()  
  
print(next(p))  
print(next(p))  
print(next(p))

即：带有yeild关键字的函数，其运行结果可以传入到next函数中，作为迭代器使用

### 3.5 模块

#### 3.5.1 模块简介

在日常程序的开发过程中，随着程序代码量越来越多，在一个文件里代码就会越来越长，最后会慢慢变得越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式例如：Java、C#、C++等。在Python中，一个.py文件就称之为一个模块（Module）。

**模块对于我们的日常开发有什么好处呢？**

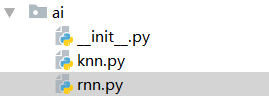
最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。

大家可能还会有些疑问，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

例：一个knn.py的文件就是一个名叫knn的模块，一个rnn.py的文件就是一个名叫rnn的模块。

假设我们的knn和rnn这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如ai，按照如下目录存放：



引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，knn.py模块的名字就变成了ai.knn，类似的，rnn.py的模块名变成了ai.rnn。

**注意:**

在Python 3.3之前，一个目录想被当成package被导入，必须包含init.py文件；而在Python 3.3及以后的版本中，\_\_init\_\_.py文件可以不需要，直接使用import后者from语法，就能直接导入目录，这样的目录称为namespace package。

换句话说，Python 3.3之后，存在两种package，一种是常规的package，即包含\_\_init\_\_.py的目录，一种是namespace package，即不包含init.py的目录。

**\_\_init\_\_.py的作用**

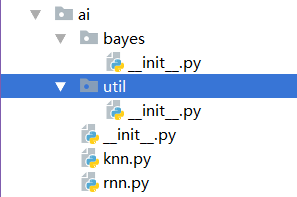
每个目录下都有\_\_init\_\_.py文件，这个是初始化模块，from-import语句导入子包时需要它，可以在里面做一些初始化工作，也可以是空文件。

注意：\_\_init\_\_.py定义的属性直接使用 顶层包.子包 的方式导入，如在目录a的\_\_init\_\_.py文件中定义init\_db()方法，调用如下：

from util import app

app.init\_db()

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：



#### 3.5.2 模块的使用

##### 3.5.2.1 import语句

有下面两种方式：

第一种：

import module1

import module2

import module3

第二种：

import module1,module2,module3

这两种方式的效果是一样的，但是第一种可读性比第二种好，推荐按照下面的顺序导入模块，并且一般在文件首部导入所有的模块：

* python标准库
* 第三方模块
* 应用程序自定义模块

也可以在函数内部导入模块，这样被导入的模块作用域是局部的

##### 3.5.2.2 from-import语句

形式上有一下两种：（**from后的参数要从项目根目录下的文件夹开始**）

单行导入

**from** ai.knn **import** demo

多行导入

**from** ai.knn **import** demo1,\  
 demo2

导入全部属性（由于容易覆盖当前名称空间中现有的名字，所以一般不推荐使用，适合模块中变量名很长并且变量很多的情况）

from module import \*

如果你不想某个模块的属性被以上方法导入，可以给该属性名称前加一个下划线(\_test)，如果需要取消隐藏，可以显示的导入该属性（from module import \_test）

除了以上导入模块中的某个变量，我们还可以使用from-import导入包中的某个模块

##### 3.5.2.3 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：name，age，PI等一系列的属性。

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；

之所以我们说，private函数和变量“不应该”其被他模块或类直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

**def** \_\_privateDemo1():  
 **return "张三"  
  
def** \_\_privateDemo2():  
 **return "李四"  
  
  
def** getting(count):  
 **if** count == 1:  
 **return** \_\_privateDemo1()  
 **else**:  
 **return** \_\_privateDemo2()

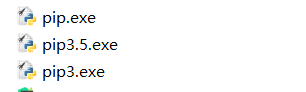
我们在模块里公开getting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用getting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

##### 3.5.2.4 第三方模块的安装

在Python中，安装第三方模块，是通过包管理工具pip来完成。

如果是Mac或Linux，安装pip（包管理工具）本身这个步骤就可以跳过了。



C:\Users\Administrator\AppData\Roaming\Tencent\Users\979641996\TIM\WinTemp\RichOle\E(127}M9JE7TC4FRRGF9)MD.png

pip install 第三方模块名称。

### 3.6 面向对象

#### 3.6.1简介

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的信息，为了表示一个学生的信息，面向过程的程序可以用一个dict表示：

stu1 = {**"name"**:**"张三"**,**"age"**:20,**"sex"**:**"男"**}  
stu2 = {**"name"**:**"李四"**,**"age"**:20,**"sex"**:**"男"**}

而处理学生信息可以通过函数实现，比如打印学生的具体信息：

**def** printMsg(stu):  
 print(**"name: %s,age: %d,sex: %s"** %(stu[**"name"**],stu[**"age"**],stu[**"sex"**]))  
printMsg(stu1)

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是Student这种数据类型应该被视为一个对象，这个对象拥有name、age和sex这三个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个printMsg消息，让对象自己把自己的数据打印出来。

**class** Student:  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex  
  
 **def** print\_msg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))  
  
stu = Student(**"莉莉"**,22,**"女"**)  
stu. print\_msg ()

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

#### 3.6.2 类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，我们拿Student类为例，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

**class** Student(object):  
 **pass**

class后面紧跟类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来（继承会在后面部分讲解），通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

如何创建实例：

stu = Student(**"莉莉"**,22,**"女"**)  
print(stu)

输出结果：

C:\Users\Administrator\AppData\Roaming\Tencent\Users\979641996\TIM\WinTemp\RichOle\CWMCNTVKL%%WTN3P@I]8{I4.png

可以看到，变量stu指向的就是一个Student的实例，后面的0x00000000028C59B0是内存地址，每个实例的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例stu绑定一个score属性：

stu.score = 98  
print(stu.score)

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，age，sex等属性绑上去：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex

注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去。

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

**数据封装**

面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例就拥有各自的name、age和sex这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的信息：

**def** print\_msg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))

但是，既然Student实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex  
  
 **def** print\_msg (self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入。

这样一来，我们从外部看Student类，就只需要知道，创建实例需要给出name和age、sex，而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给Student类增加新的方法，比如get\_lastname：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex  
  
 **def** print\_msg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))  
  
 **def** get\_lastname(self):  
 **return** self.name[0]  
  
stu = Student(**"张莉莉"**,22,**"女"**)  
print(stu.get\_lastname())

同样get\_lastname()方法可以直接在实例变量上调用，不需要知道内部实现细节。

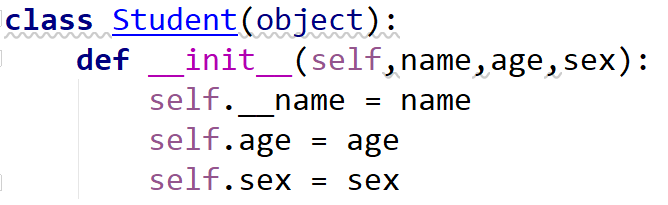
#### 3.6.3 访问限制

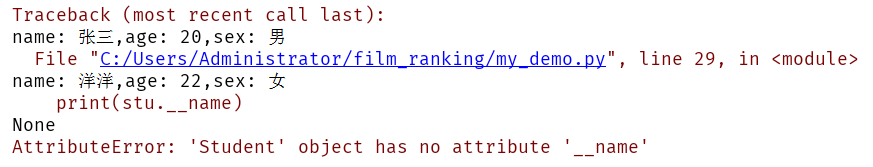
在class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、age、sex属性：

stu = Student(**"张莉莉"**,22,**"女"**)  
print(stu.name)  
print(stu.age)  
print(stu.sex)

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：





这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name怎么办？可以给Student类增加get\_name这样的方法。

**def** get\_name(self):  
 **return** self.\_\_name

如果这时候想修改name属性值,那么增加set\_name（）方法即可。

**def** set\_name(self,name):  
 self.\_\_name = name

你也许会问，原先那种直接通过stu.age = 99也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数。

例如：人类目前的年龄不能可能大于200岁。

注意：

在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_age\_\_这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

stu = Student(**"张莉莉"**,22,**"女"**)  
print(stu.\_Student\_\_name)  
print(stu.age)  
print(stu.sex)

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

最后注意下面的这种错误写法：

stu = Student(**"张莉莉"**,22,**"女"**)  
print(stu.get\_name())  
stu.\_\_name = **"小臭鱼"**print(stu.get\_name())

表面上看，外部代码“成功”地设置了\_\_name变量，但实际上这个\_\_name变量和class内部的\_\_name变量不是一个变量！内部的\_\_name变量已经被Python解释器自动改成了\_Student\_\_name，而外部代码给stu新增了一个\_\_name变量。不信试试

#### 3.6.4 继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。对于继承object父类，在Python3版本中，不需要显式的声明

比如，我们已经编写了一个名为Cat的class，有一个run()方法可以直接打印：

**class** Car(object):  
 **def** run(self):  
 print(**"汽车在跑！！！！"**)

当我们需要编写Bmw和Jaguar类时，就可以直接从Cat类继承：

**class** Car:  
 **def** run(self):  
 print(**"汽车在跑"**)  
  
**class** Bmw(Car):  
 **def** run(self):  
 print(**"Bwm在跑"**)  
  
**class** Jaguar(Car):  
 **def** run(self):  
 print(**"Jaguar在跑"**)  
  
bmw = Bmw()  
bmw.run()

这时候实际是调用的Car类的run函数。

当子类和父类都存在相同的run()方法时，我们说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

b = Car()  
c = Bmw()  
  
print(isinstance(b,Car))  
print(isinstance(c,Bmw))  
**'''可是尝试一下'''**print(isinstance(c,Car))

True

True

True

True

看来c不仅仅是Bmw，c还是Car。

不过仔细想想，这是有道理的，因为Bmw是从Car继承下来的，当我们创建了一个Bmw的实例c时，我们认为c的数据类型是Bmw没错，但c同时也是Car也没错，Bmw本来就是Car的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

print(isinstance(b,Bmw))

False

Bmw可以看成Car，但Car不可以看成Bmw。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个Car类型的变量：

**class** Car(object):  
 **def** run(self):  
 print(**"汽车在跑！！！！"**)  
  
**class** Bmw(Car):  
 **def** run(self):  
 print(**"BMW在跑！！！！"**)  
  
**class** Jaguar(Car):  
 **def** run(self):  
 print(**"Jaguar在跑！！！！"**)  
  
  
**def** run\_demo(car):  
 car.run()  
  
run\_demo(Car())  
run\_demo(Bmw())  
run\_demo(Jaguar())

运行结果：

汽车在跑！！！！

BMW在跑！！！！

Jaguar在跑！！！！

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Benz类型，也从Car派生。

你会发现，新增一个Car的子类，不必对run\_demo()做任何修改，实际上，任何依赖Car作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

多态的好处就是，当我们需要传入Bmw、Jaguar、Tortoise……时，我们只需要接收Car类型就可以了，因为Bmw、Jaguar、Benz……都是Car类型，然后，按照Car类型进行操作即可。由于Car类型有run()方法，因此，传入的任意类型，只要是Car类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Car类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Car、Bmw、Jaguar还是Benz对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Car的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Car子类；

对修改封闭：不需要修改依赖Car类型的run\_demo()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：

图略。。。。。。。。。。。。。。。。

**静态语言 & 动态语言**

对于静态语言（例如Java、C++、C）来说，如果需要传入Car类型，则传入的对象必须是Car类型或者它的子类，否则，将无法调用run()方法，也可以说是在编译器就要确定类型。而对于Python这样的动态语言来说，则不一定需要传入Car类型。我们只需要保证传入的对象有一个run()方法就可以了：

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系。“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”在鸭子类型中，关注点在于对象的行为，能作什么；而不是关注对象所属的类型。

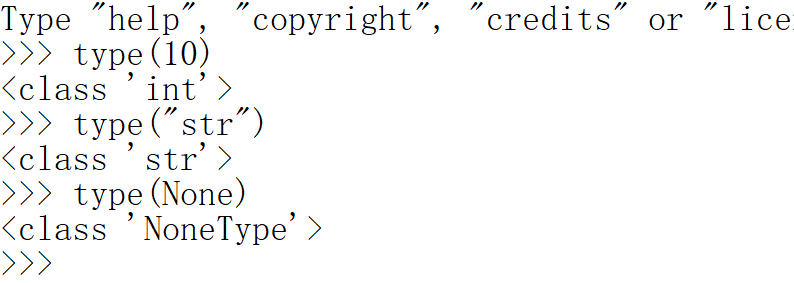
Python的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个read()方法，返回其内容。但是，许多对象，只要有read()方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了read()方法的对象。

#### 3.6.5 获取对象信息

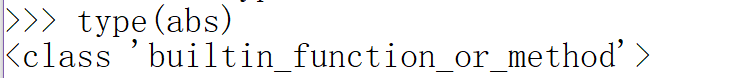
当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

##### type()函数

基本类型都可以用type()判断：



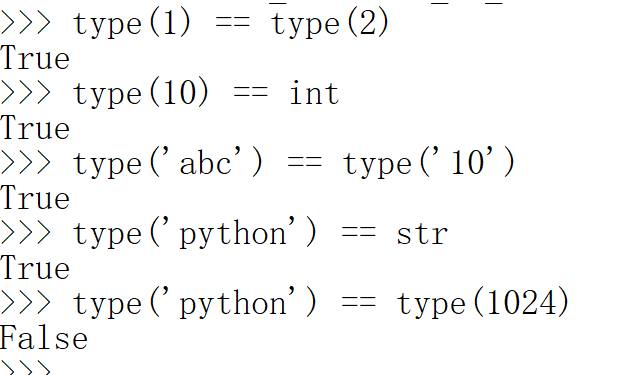
如果一个变量指向函数或者类，也可以用type()判断：



**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.\_\_name = name  
 self.age = age  
 self.sex = sex  
 **def** get\_name(self):  
 **return** self.\_\_name  
 **def** set\_name(self,name):  
 self.\_\_name = name  
 **def** printMsg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))  
  
 **def** get\_lastname(self):  
 **return** self.name[0]  
  
stu = Student(**"张莉莉"**,22,**"女"**)  
print(type(stu))

<class '\_\_main\_\_.Student'>

但是type()函数返回的是什么类型呢？它返回对应的Class类型。如果我们要在if语句中判断，就需要比较两个变量的type类型是否相同：



如果想判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

**import** types  
  
**def** func():  
 **pass**print(type(func) == types.FunctionType)  
print(type(abs) == types.BuiltinFunctionType) #built-in type(内置类型)与 user-defined type(用户自定义类型)相对应

print(type(**lambda** x: x) == types.LambdaType) #Lambda表达式类型  
print(type((x **for** x **in** range(10))) == types.GeneratorType)

运行结果：

True

True

True

True

##### isinstance()函数

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。

如果继承关系是：

object -> Car -> Volkswagen -> Porsche

那么，isinstance()就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

a = Car()  
d = Volkswagen()  
c = Porsche()

isinstance(c, Porsche)  
isinstance(c, Volkswagen)  
isinstance(c, Car)

运行结果为：True True True

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

print(isinstance([1, 2, 3], (list, tuple)))  
print(isinstance((1, 2, 3), (list, tuple)))

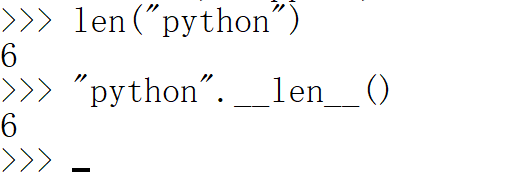
##### dir()函数

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

print(dir(**"Python"**))

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：



我们自己写的类，如果也想用len(obj)的话，就自己写一个\_\_len\_\_()方法：

**class** MyObj(object):  
 **def** \_\_len\_\_(self):  
 **return** 10  
  
obj = MyObj()  
print(len(obj))

##### getattr()、setattr()以及hasattr()函数

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

**class** MyObj(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"莉莉"  
 def** \_\_len\_\_(self):  
 **return** 10  
  
obj = MyObj()  
print(hasattr(obj,**"name"**))  
**if** hasattr(obj,**"age"**):  
 print(obj.age)  
**else**:  
 setattr(obj,**"age"**,20)  
print(getattr(obj,**"age"**,**"没有该属性"**)) *#这里最好添加个默认值，否则没有该属性会报错。*

**使用getattr()可以获得对象的方法：**

**class** MyObj(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"莉莉"  
 def** \_\_len\_\_(self):  
 **return** 10  
 **def** run(self,content):  
 print(content)  
  
obj = MyObj()  
func = getattr(obj,**"run"**)  
func(**"获取方法！！！"**)

#### 3.6.6 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

例：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.\_\_name = name  
 self.age = age  
 self.sex = sex

stu = Student(**"莉莉"**,22,**"女"**)  
stu.score = 100

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

**class** Student(object):  
 name = **"张三"**

**当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。**

stu = Student()  
print(stu.name)# 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性  
print(Student.name)

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

**class** Student(object):  
 name = **"张三"  
 def** \_\_init\_\_(self):  
 self.name = **"李四"**stu = Student()  
print(Student.name)  
print(stu.name)# 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性  
**del** stu.name  
print(stu.name)

另外类方法和静态方法，分别使用@classmethod和@staticmethod修饰

### 3.7文件IO

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

#### 3.7.1文件读写

Python内置了读写文件的函数，用法和C语言是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

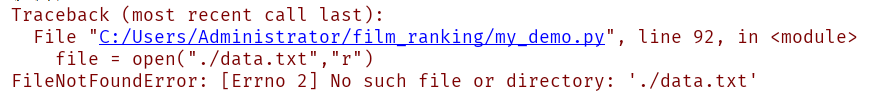
**读文件：**

要以读文件的模式打开一个文件对象，使用Python内置的open()函数，传入文件名和标示符：

file = open(**"./data.txt"**,**"r"**)

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，open()函数就会抛出一个IOError的错误，并且给出错误码和详细的信息告诉你文件不存在：



如果文件打开成功，接下来，调用read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示：

file = open(**"./data.txt"**,**"r"**)  
str = file.read()  
print(str)

最后一步是调用close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

file.close()

由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

关于异常处理的格式：

**try**:  
 print(**"业务部分"**)

**except** Exception **as** result:  
 print(result)  
 print(**"继续"**)  
**finally**:  
 print(**"最终总是要执行"**)

例：

**try**:  
 file = open(**"./data.txt"**,**"r"**)  
 str = file.read()  
 print(str)  
**finally**:  
 **if** file:  
 file.close()

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

**with** open(**'./data.txt'**, **'r'**) **as** file:  
 print(file.read())

这和前面的try ... finally是一样的，但是代码更加简洁，并且不必调用f.close()方法。

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list。因此，要根据需要决定怎么调用。

如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用read(size)比较保险；如果是配置文件，调用readlines()最方便：

file = open(**"./data.txt"**,**"r"**)  
**for** line **in** file.readlines():  
 print(line.strip())

**file-like Object**

像open()函数返回的这种有个read()方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个read()方法就行。

StringIO就是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件即可：

file = open(**"1.png"**,**"rb"**)  
print(file.read())*# 十六进制表示的字节*

**字符编码**

要读取非UTF-8编码的文本文件，需要给open()函数传入encoding参数，例如，读取GBK编码的文件：

*#data使用Windows系统创建，默认为gbk*

file = open(**"data.txt"**,**"r"**,encoding=**"gbk"**)  
print(file.read())

遇到有些编码不规范的文件，你可能会遇到UnicodeDecodeError，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况，open()函数还接收一个errors参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

file = open(**"data.txt"**,**"r"**,encoding=**"gbk"**,errors=**"ignore"**)  
print(file.read())

**写文件**

写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件：

file = open(**"data.txt"**,**"w"**)  
file.write(**"Tensorflow"**)  
file.close()

你可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用close()的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用with语句来得保险：

**with** open(**"data.txt"**,**"a"**) **as** file:  
 file.write(**"tensorflow"**)

要写入特定编码的文本文件，请给open()函数传入encoding参数，将字符串自动转换成指定编码。

细心的童鞋会发现，以'w'模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果我们希望追加到文件末尾怎么办？可以传入'a'以追加（append）模式写入。

**open mode**

|  |  |
| --- | --- |
| w | 以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容 |
| a | 以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建 |
| r+ | 以读写方式打开文件，可对文件进行读和写操作。 |
| w+ | 消除文件内容，然后以读写方式打开文件。 |
| a+ | 以读写方式打开文件，并把文件指针移到文件尾。 |
| b | 以二进制模式打开文件，而不是以文本模式。该模式只对Windows或Dos有效，类Unix的文件是用二进制模式进行操作的。 |

**操作文件对象方法**

|  |  |
| --- | --- |
| f.close() | 关闭文件，记住用open()打开文件后一定要记得关闭它，否则会占用系统的可打开文件句柄数。 |
| f.fileno() | 获得文件描述符，是一个数字 |
| f.flush() | 刷新输出缓存 |
| f.isatty() | 如果文件是一个交互终端，则返回True，否则返回False。 |
| f.read([count]) | 读出文件，如果有count，则读出count个字节。 |
| f.readline() | 读出一行信息。 |
| f.readlines() | 读出所有行，也就是读出整个文件的信息。 |
| f.seek(offset[,where]) | 把文件指针移动到相对于where的offset位置。where为0表示文件开始处，这是默认值 ；1表示当前位置；2表示文件结尾。 |
| f.tell() | 获得文件指针位置。 |
| f.truncate([size]) | 截取文件，使文件的大小为size。 |
| f.write(string) | 把string字符串写入文件。 |
| f.writelines(list) | 把list中的字符串一行一行地写入文件，是连续写入文件，没有换行。 |

#### 3.7.2 StringIO和BytesIO

##### StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。

StringIO顾名思义就是在内存中读写str。

要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可：

**from** io **import** StringIO  
  
file = StringIO(**"Hello \n Python"**)  
**for** line **in** file.readlines():  
 print(line.strip())

**from** io **import** StringIO  
  
str = StringIO()  
str.write(**"Hello\n"**)  
str.write(**"Python"**)  
str.flush()  
print(str.getvalue())  
str.close()

##### BytesIO

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些bytes：

**from** io **import** BytesIO  
  
bt = BytesIO()  
bt.write(**"大家好python"**.encode(**"utf-8"**))  
bt.flush()  
print(bt.getvalue())  
bt.close()

请注意，写入的不是str，而是经过UTF-8编码的bytes。

和StringIO类似，可以用一个bytes初始化BytesIO，然后，像读文件一样读取：

**from** io **import** BytesIO  
  
bt = BytesIO(**b"\xe5\xa4\xa7\xe5\xae\xb6\xe5\xa5\xbdpython"**)  
print(bt.read().decode(**"utf-8"**))

#### 3.7.3 操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如dir、copy等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的os模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用os模块的基本功能：

**import** os  
print(os.name)*#操作系统类型*

如果是posix，说明系统是Linux、Unix或Mac OS X，如果是nt，就是Windows系统。

要获取详细的系统信息，可以调用uname()函数：

os.uname()

注意：

uname()函数在Windows上不提供，也就是说，os模块的某些函数是跟操作系统相关的。

**环境变量**

print(os.environ)

运行结果有点多

environ({'ALLUSERSPROFILE': 'C:\\ProgramData', 'APPDATA': 'C:\\Users\\Administrator\\AppData\\Roaming', 'COMMONPROGRAMFILES': 'C:\\Program Files\\Common Files', 'COMMONPROGRAMFILES(X86)': 'C:\\Program Files (x86)\\Common Files', 'COMMONPROGRAMW6432': 'C:\\Program Files\\Common Files', 'COMPUTERNAME': 'USER-20170627XN', 'COMSPEC': 'C:\\Windows\\system32\\cmd.exe', 'FP\_NO\_HOST\_CHECK': 'NO', 'HOMEDRIVE': 'C:', 'HOMEPATH': '\\Users\\Administrator', 'JAVA\_HOME': 'D:\\Java\\jdk1.8', 'LOCALAPPDATA': 'C:\\Users\\Administrator\\AppData\\Local', 'LOGONSERVER': '\\\\USER-20170627XN', 'NUMBER\_OF\_PROCESSORS': '4', 'NVIDIAWHITELISTED': '0x01', 'OS': 'Windows\_NT', 'PATH': 'C:\\ProgramData\\Oracle\\Java\\javapath;C:\\Windows\\system32;C:\\Windows;C:\\Windows\\System32\\Wbem;C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\;C:\\Program Files (x86)\\NVIDIA Corporation\\PhysX\\Common;D:\\TortoiseSVN\\bin;D:\\VisualSVN Server\\bin;D:\\Git\\cmd;D:\\nodejs\\;D:\\Microsoft VS Code\\bin;D:\\Python\\Python36\\Scripts\\;D:\\Python\\Python36\\;D:\\Microsoft VS Code\\bin;C:\\Users\\Administrator\\AppData\\Local\\GitHubDesktop\\bin;D:\\mysql5.7\\bin;D:\\Sencha\\Cmd\\6.5.2.15;C:\\Users\\Administrator\\AppData\\Local\\atom\\bin;C:\\Users\\Administrator\\AppData\\Roaming\\npm;D:\\Java\\jdk1.8\\bin;D:\\MySQL\\MySQL Server 5.7\\bin;', 'PATHEXT': '.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC', 'PROCESSOR\_ARCHITECTURE': 'AMD64', 'PROCESSOR\_IDENTIFIER': 'Intel64 Family 6 Model 42 Stepping 7, GenuineIntel', 'PROCESSOR\_LEVEL': '6', 'PROCESSOR\_REVISION': '2a07', 'PROGRAMDATA': 'C:\\ProgramData', 'PROGRAMFILES': 'C:\\Program Files', 'PROGRAMFILES(X86)': 'C:\\Program Files (x86)', 'PROGRAMW6432': 'C:\\Program Files', 'PSMODULEPATH': 'C:\\Windows\\system32\\WindowsPowerShell\\v1.0\\Modules\\;D:\\VisualSVN Server\\PowerShellModules', 'PUBLIC': 'C:\\Users\\Public', 'PYCHARM\_HOSTED': '1', 'PYTHONIOENCODING': 'UTF-8', 'PYTHONPATH': 'C:\\Users\\Administrator\\film\_ranking', 'PYTHONUNBUFFERED': '1', 'SESSIONNAME': 'Console', 'SHIM\_MCCOMPAT': '0x810000001', 'SYSTEMDRIVE': 'C:', 'SYSTEMROOT': 'C:\\Windows', 'TEMP': 'C:\\Users\\ADMINI~1\\AppData\\Local\\Temp', 'TMP': 'C:\\Users\\ADMINI~1\\AppData\\Local\\Temp', 'USERDOMAIN': 'USER-20170627XN', 'USERNAME': 'Administrator', 'USERPROFILE': 'C:\\Users\\Administrator', 'VBOX\_MSI\_INSTALL\_PATH': 'D:\\Oracle\\VirtualBox\\', 'VISUALSVN\_SERVER': 'D:\\VisualSVN Server\\', 'VS140COMNTOOLS': 'C:\\Program Files (x86)\\Microsoft Visual Studio 14.0\\Common7\\Tools\\', 'WINDIR': 'C:\\Windows', 'WINDOWS\_TRACING\_FLAGS': '3', 'WINDOWS\_TRACING\_LOGFILE': 'C:\\BVTBin\\Tests\\installpackage\\csilogfile.log', '\_DFX\_INSTALL\_UNSIGNED\_DRIVER': '1'})

要获取某个环境变量的值：

print(os.environ.get(**"PATH"**))

##### 操作文件和目录

操作文件和目录的函数一部分放在os模块中，一部分放在os.path模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

**import** os  
*# 查看当前目录的绝对路径*print(os.path.abspath(**'./'**))  
*# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来*print(os.path.join(**"D:\Just-Do-IT"**,**"python"**))  
*#创建一个目录*os.mkdir(**"D:/Just-Do-IT/python"**)  
*# 删掉一个目录*os.rmdir(**"D:/Just-Do-IT/python"**)

把两个路径合成一个时，不要直接拼字符串，而要通过os.path.join()函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，os.path.join()返回这样的字符串：

part-1/part-2

而Windows下会返回这样的字符串：

part-1\part-2

拆分路径：

print(os.path.split(**"D:/Just-Do-IT/python"**))

要拆分路径时，不要直接去拆字符串，而要通过os.path.split()函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名。

获取文件拓展名：

os.path.splitext(**"D:/Just-Do-IT/conf.txt"**)

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个data.txt文件：

重命名与删除文件

os.rename(**"data.txt"**,**"data.py"**)

os.remove(**"data.py"**)

但是复制文件的函数居然在os模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是shutil模块提供了copyfile()的函数，你还可以在shutil模块中找到很多实用函数，它们可以看做是os模块的补充。

1. shutil.copy(fsrc, fdst[, length=16\*1024])    copy文件内容到另一个文件，不能copy到同一文件夹中，可以copy指定大小的内容，该方法只能copy文件，不能copy文件夹，操作完成后返回新创建文件的路径
2. shutil.copyfile(src, dst)：从源src复制到dst中去，前提是目标地址是具备可写权限。抛出的异常信息为IOException. 如果当前的dst已存在的话就会被覆盖掉。只是会复制其权限其他的东西是不会被复制的。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

*#获取c盘下所有的文件夹*

l = [x **for** x **in** os.listdir(**'c:/'**) **if** os.path.isdir(os.path.join(**'c:/'**,x))]  
print(l)

*#输出当前文件夹下的py文件*

[x **for** x **in** os.listdir(**'.'**) **if** os.path.isfile(x) **and** os.path.splitext(x)[1] ==**'.py'**]

#### 3.7.4 序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

d = dict(name=**"Tom"**,age=20,sex=**"男"**)

可以随时修改变量，比如把name改成'Bill'，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的'Bill'存储到磁盘上，下次重新运行程序，变量又被初始化为'Tom'。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了pickle模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

**import** pickle  
  
d = dict(name=**"Tom"**,age=20,sex=**"男"**)  
print(pickle.dumps(d))

pickle.dumps()方法把任意对象序列化成一个bytes，然后，就可以把这个bytes写入文件。或者用另一个方法pickle.dump()直接把对象序列化后写入一个file-like Object：

**import** pickle  
  
d = dict(name=**"Tom"**,age=20,sex=**"男"**)  
file = open(**"dump.txt"**,**"wb"**)  
pickle.dump(d,file)  
file.close()

看看写入的dump.txt文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个bytes，然后用pickle.loads()方法反序列化出对象，也可以直接用pickle.load()方法从一个file-like Object中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

file = open(**"dump.txt"**,**"rb"**)  
d = pickle.load(file)  
file.close()  
print(d)

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

##### Json

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

|  |  |
| --- | --- |
| JSON类型 | Python类型 |
| {} | Dict |
| [] | list |
| “string” | str |
| 1234.56 | Int/flaot |
| True/false | True/False |
| Null | None |

Python内置的json模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

**import** json  
d = dict(name=**"Tom"**,age=20,sex=**"男"**)  
print(json.dumps(d))

dumps()方法返回一个str，内容就是标准的JSON。类似的，dump()方法可以直接把JSON写入一个file-like Object。

要把JSON反序列化为Python对象，用loads()或者对应的load()方法，前者把JSON的字符串反序列化，后者从file-like Object中读取字符串并反序列化：

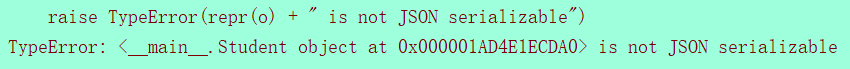
str = **'{"age": 20, "sex": "男", "name": "Tom"}'**print(json.loads(str))

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的str与JSON的字符串之间转换。

##### Json进阶

Python的dict对象可以直接序列化为JSON的{}，不过，很多时候，我们更喜欢用class表示对象，比如定义Student类，然后序列化：

**import** json  
**class** Student(object):  
 **def** \_\_init\_\_(self, name, age, score):  
 self.name = name  
 self.age = age  
 self.score = score  
  
s = Student(**'Tom'**, 20, 88)  
print(json.dumps(s))



错误的原因是Student对象不是一个可序列化为JSON的对象。

如果连class的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看dumps()方法的参数列表，可以发现，除了第一个必须的obj参数外，dumps()方法还提供了一大堆的可选参数。

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把Student类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Student实例变为一个JSON的{}对象。

可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Student专门写一个转换函数，再把函数传进去即可：

**class** Student(object):  
 **def** \_\_init\_\_(self, name, age, score):  
 self.name = name  
 self.age = age  
 self.score = score  
  
**def** student2dict(std):  
 **return** {  
 **'name'**: std.name,  
 **'age'**: std.age,  
 **'score'**: std.score  
 }  
  
s = Student(**'Tom'**, 20, 88)  
print(json.dumps(s,default=student2dict))

不过，下次如果遇到一个Teacher类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意class的实例变为dict：

print(json.dumps(s, default=**lambda** obj: obj.\_\_dict\_\_))

因为通常class的实例都有一个\_\_dict\_\_属性，它就是一个dict，用来存储实例变量。也有少数例外，比如定义了\_\_slots\_\_的class。

同样的道理，如果我们要把JSON反序列化为一个Student对象实例，loads()方法首先转换出一个dict对象，然后，我们传入的object\_hook函数负责把dict转换为Student实例：

**def** dict2student(d):  
 **return** Student(d[**'name'**], d[**'age'**], d[**'score'**])  
json\_str = **'{"age": 20, "score": 88, "name": "Bob"}'**print(json.loads(json\_str, object\_hook=dict2student))

