python 学习笔记

hzzmail[1]

2017-08-14

[1]hzzmail@163.com

Chapter 1

python 重要注意点

1. py 脚本的命名

   py 脚本的命名有时会产生干扰，所以要需要注意。比如 signal.py 这样一个命名可能会出现错误，因为很有可能某些模块中具有这样的命名的子模块。如果把自编的脚本命名为该名字，就会出错。这种情况如果用 ipython 执行可能会通过，但如果用 python 就会出错，因为 ipython 不搜索当前工作目录，而 python 搜索模块首先搜索当前目录。

   因此，对于自编脚本，要尽可能有区分度，不要使其与标准模块或者其它安装模块的文件名相同。比如加个 test，加个 user 等

2. 运行脚本

   直接在 dos 下或 win 下运行 python file.py 即可

   调用 python 解释器，则直接在 dos 下输入命令 python，退出则按 ctrl+z 即可

3. Python 是强类型的

   您有一个整数，如果不明确地进行转换，不能将把它当成一个字符串。

4. print 函数用法

   Python 3.6 版本的 print 函数需要把打印内容用括号括起来，这与之前 2.x 版本是不一样的。比如: print ('python 基本原则:') 而不是: print 'python 基本原则:'

   还要注意格式化中的间隔符是%

   如果要去打印不换行，则使用参数 end='' 比如: print ('python 基本原则:',end='')

5. 暂停代码用于简单调试

```
1        print("input␣a␣key␣to␣continue:")
2        vtemp=input()
```

6. 万物皆对象，其实一切也都是指针

   函数也是对象，可以进行非直接调用

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

def func(x,y,z):return x+y+z
print(func(1,2,3))

f=lambda x,y,z:x+y+z
print(f(1,2,3))

x=func
print(x(2,3,4))

def addsth(x,y):
    return x+y

def indirect(fun,arg1,arg2):
    return fun(arg1,arg2)
print(indirect(addsth,3,4))

def fun(x,y,z):
    return x+y+z

def indirect(fun,arg1,arg2,arg3):
    return fun(arg1,arg2,arg3)
print(indirect(func,3,4,5))
```

7. 帮助和用法

   object.___doc___ 是对象的 doc string 文本内容

   dir(object) 查对象属性，方法

   help(object) 查对象帮助

8. 模块导入，一个 py 文件就是一个模块

   在 Python 中的 import 就像 Perl 中的 require。import 一个 Python 模块后，您就可以使用 module.function 来访问它的函数；require 一个 Perl 模块后，您就可以使用 module::function 来访问它的函数。

   注意: 在脚本的当前目录下不要存在于 python 其它标准库同名的文件，否则加载该库就会失败，因为它去加载当前目录下的同名文件。比如当前脚本 test.py 的同目录下有一个 copy.py 文件，那么 test 中使用 copy 模块时就会出现问题。

   内置属性 ___name___: 如果 import 模块，那么 ___name___ 的值通常为模块的文件名，不带路径或者文件扩展名。但是您也可以像一个标准的程序一样直接运行模块，在这种

情况下 ___name___ 的值将是一个特别的缺省值，___main___。(就可以在模块内部为您的模块设计一个测试套件，在其中加入这个 if 语句。当您直接运行模块，___name___ 的值是 ___main___，所以测试套件执行。当您导入模块，___name___ 的值就是别的东西了，所以测试套件被忽略。这样使得在将新的模块集成到一个大程序之前开发和调试容易多)

9. 搜索路径

当导入一个模块时，Python 在几个地方进行搜索。明确地，它会对定义在 sys.path 中的目录逐个进行搜索。它只是一个 list (列表)，您可以容易地查看它或通过标准的 list 方法来修改它。比如:sys.path.append('mypath')

```
>>> import sys
>>> sys.path
['D:\\d_test\\work_python', 'C:\\Python36-32\\python36.zip', 'C:\\Python36-32\\DLLs', 'C:\\
    Python36-32\\lib', 'C:\\Python36-32', 'C:\\Python36-32\\lib\\site-packages']
>>>
```

10. 代码缩进

Python 函数没有明显的 begin 和 end，没有标明函数的开始和结束的花括号。唯一的分隔符是一个冒号 (:)，接着代码本身是缩进的。Python 使用硬回车来分割语句，冒号和缩进来分割代码块。C++ 和 Java 使用分号来分割语句，花括号来分割代码块。

11. 赋值生成引用而不是拷贝

比如 L=[1,2,3] M=[1,L,3] 给 L 赋值的列表不仅被 L 引用，也被 M 引用。这是共享引用问题。可以这么理解，当一个对象构成一个复合对象时，比如这里的 L 构成 M 时，直接使用 L 就产生共享引用问题，当 L 引用的对象修改时，M 也会跟着修改。当用全切片 L[:] 代替 L 时则 M 不会发生修改。所以构成复合对象的时候一定要注意，这个与浅复制，深复制有点相似。对于浅复制和深复制也要注意，两者的差别在于浅复制是在生成新的对象时，会找原对象中的元素，某个元素是一个对象那么就会复制该对象的引用。

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_


import copy

print(dir(copy))

L=[1,2,3]
M=["X",L,"Y"]#其中的L，插入的是一个引用
print(M)
L[1]=0
```

```
13      print(M)
14      M=["X",L[:],"Y"]#其中L[:]，插入的是L的一个完整复制
15      print(M)
16      L[1]=2
17      print(M)
18
19      X=L*4
20      Y=[L]*4#其中的L，插入的是一个引用
21      Z=[L[:]]*4#其中L[:]，插入的是L的一个完整复制
22      print(X)
23      print(Y)
24      print(Z)
25      L[1]=0
26      print(X)
27      print(Y)
28      print(Z)
29
30      print("\n")
31      L=[1,2,3]
32      M=["X",L,"Y"]#其中的L，插入的是一个引用，因此L和M的修改是相关的
33      N=copy.copy(M)#因为N是M的浅复制，所以N和M中对象也是相关的
34      S=copy.deepcopy(M)#但对于深复制，则不相关
35      print(L)
36      print(N)
37      print(S)
38      L[1]=0
39      print(L)
40      print(N)
41      print(S)
42      M[1][1]=5
43      print(L)
44      print(N)
45      print(S)
46
47      print("\n")
48      L=[1,2,3]
49      M=["X",L[:],"Y"]#其中L[:]，插入的是L的一个完整复制，因此L和M的修改已经不相关
50      N=copy.copy(M)
51      S=copy.deepcopy(M)
52      print(L)
53      print(N)
54      print(S)
55      L[1]=0
56      print(L)
57      print(N)
58      print(S)
59      M[1][1]=5
60      print(L)
61      print(N)
62      print(S)
```

```
63      M[0]="z"
64      print(N)
65      print(S)
```

12. python 程序打包成 exe 文件。

    工具主要有:py2exe，pyinstaller，具体用法百度即可。

1.1 语言特性

数据类型，常量和变量

基本数据结构 (dict):

- 赋值是引用还是新创建一个对象？

对象操作符 (标准，基类)，函数 (标准，工厂)，方法 (标准，内建)

流程控制 (分支，循环)

函数与过程:

- 不同作用范围的数据 (全局和局部) 的使用
- 数据的传递方式: 传值还是传址，还是传的引用？
- 返回的方式:return，还是利用带的参数，返回的是什么类型的数据，带的参数怎么个传递法

Chapter 2

python 入门经典

参考:python 入门经典，该书的模块，类和异常等问题还可以再细看一下。

python 可以分为模块，语句和对象。程序有模块组成，模块中包含语句，语句生成并处理对象。使用 c 或 c++ 这一的低级语言，大多数的工作都集中在对象的实现上 (也就是有些人说的数据结构) 以表示自己应用领域的组件。需要设计内存结构，管理内存分配，实现查找和访问例程等。这些事听起来烦人且容易出错，并经常会让你偏离真正的编程目标。在典型的 python 程序中，大多数这些烦心事都没有了。因为 python 提供了功能强大的对象类型，作为语言的一部分，在你开始解决问题之前不需要为对象的实现书写代码。

函数是一种通用的程序结构化工具，简短的来说，函数有两个作用: 1. 代码的重用，对于要在不同地方，不同时间，不止一次使用的代码，函数是最简单的逻辑打包方式。2. 过程的分解，函数是把系统分割成许多片段的一种工具，每个函数都有一个定义良好的作用。

模块是提供了一种简单的方法，可以把组件组织成一个系统。从抽象的角度看，模块至少有三个作用: 1. 代码重用

2. 系统名字空间的划分。模块是 python 最高级的程序组织单位，我们看到 python 中的东西都是在模块里面的，执行的代码，生成的对象总是隐含地包含在一个模块中。因此，模块是天然的组织系统组件的工具。

3. 实现服务或数据共享。从功能的角度看，模块也用于在整个系统里实现组件共享，这需要一个拷贝。如果你要提供一个可以在多个函数中使用的全局数据结构，你可以将其写在一个模块文件中，再被许多客户导入。

2.1 一些内置函数和标准模块

dir 函数返回任意对象的属性和方法列表，包括模块对象、函数对象、字符串对象、列表对象、字典对象

dir(___builtins___) 可以查看内置函数包括:

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError
    ', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', '
    ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', '
    EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', '
    FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', '
    IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', '
    MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', '
    NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', '
    ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', '
    RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', '
    SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
    'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', '
    ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '_', '__build_class__', '__debug__', '
    __doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii',
    'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', '
    credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', '
    format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', '
    isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', '
    next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', '
    round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
    'zip']
```

2.1.1 转换，数字，比较

```
print( '数据类型转换:' )
print (int('123'))
print (int(13.56))
print (float('103.7'))
#print (long('103.7')) #python3已结没有长整数这一概念
print (str(1.25))
print (('\u0100'))
print (bool(1))
print (bool(0))
print (bool(''))


print(str(dir()))#返回任何对象的字符串表示
print(list("tomato"))#返回一个序列的对应列表
print(tuple("tomato"))#返回一个序列的对应元组
print(tuple([0]))#
print(int("3"))#把一个字符串或数字转换成整数
#print(int("3.1"))#报错
print(int(3.123))
print(float("3.123456"))#把一个字符串或数字转换成浮点数
print(complex(1,2))#创建一个real+imag*j的复数
print(hex(100))#把一个整数转换为16进制数字符串
#print(hex("100"))#报错，字符串不行
print(oct(100))#把一个整数转换为8进制数字符串
```

```
print(bin(100))#把一个整数转换为2进制数字符串
print (ord('a'))#返回一个单字符的unicode码
print (chr(97))#返回一个unicode码对应的单字符
print (min([5,1,2,4,3]))#返回非空序列的最小成员
print (min(5,1,2,4,3))#返回非空序列的最小成员
print (min('pif','paf','pof'))#返回非空序列的最小成员
print (min('pifzet'))#返回非空序列的最小成员
print (max([5,1,2,4,3]))#返回非空序列的最大成员
print (max(5,1,2,4,3))#返回非空序列的最小成员
print (max('pifzet'))#返回非空序列的最小成员

#字符串是字符的序列所以可以用map
lsta=map(ord,"text")
print(lsta)
lstb=map(chr,lsta)
print(lstb)
print("".join(lstb))
```

2.1.2 属性操作函数

```
>>> hasattr.__doc__
'Return whether the object has an attribute with the given name.\n\nThis is done by calling getattr(obj, name) and catching AttributeError.'
hasattr(object,attributename) 如果object有属性attributename，则返回1，否则返回0
>>> getattr.__doc__
"getattr(object, name[, default]) -> value\n\nGet a named attribute from an object; getattr(x, 'y') is equivalent to x.y.\nWhen a default argument is given, it is returned when the attribute doesn't\nexist; without it, an exception is raised in that case."
getattr(object,attributename[,default]) 返回object有属性attributename属性，如果属性不存在，若指定了缺省值default就返回缺省值，否则引发attributeerror异常
>>> setattr.__doc__
"Sets the named attribute on the given object to the specified value.\n\nsetattr(x, 'y', v) is equivalent to ``x.y = v''"
setattr(object,attributename,value]) 把object的属性attributename赋值为value，如果不支持属性创建和修改，则引发typeerror异常
>>> delattr.__doc__
"Deletes the named attribute from the given object.\n\ndelattr(x, 'y') is equivalent to ``del x.y''"
delattr(object,attributename,value]) 删除object的属性attributename赋值为value，如果属性不存在，则引发attributeerror异常
```

2.1.3 执行程序函数

import 引入模块

```
>>> exec.__doc__
'Execute the given source in the context of globals and locals.\n\nThe source may be a string representing one or more Python statements\nor a code object as returned by compile().\nThe globals
```

```
    must be a dictionary and locals can be any mapping,\ndefaulting to the current globals and locals
    .\nIf only globals is given, locals defaults to it.'
>>> compile.__doc__
"Compile source into a code object that can be executed by exec() or eval().\n\nThe source code may
    represent a Python module, statement or expression.\nThe filename will be used for run-time error
    messages.\nThe mode must be 'exec' to compile a module, 'single' to compile a\nsingle (interactive)
    statement, or 'eval' to compile an expression.\nThe flags argument, if present, controls which
    future statements influence\nthe compilation of the code.\nThe dont_inherit argument, if true,
    stops the compilation inheriting\nthe effects of any future statements in effect in the code
    calling\ncompile; if absent or false these statements do influence the compilation,\nin addition to
    any features explicitly specified."
>>> eval.__doc__
'Evaluate the given source in the context of globals and locals.\n\nThe source may be a string
    representing a Python expression\nor a code object as returned by compile().\nThe globals must be a
    dictionary and locals can be any mapping,\ndefaulting to the current globals and locals.\nIf only
    globals is given, locals defaults to it.'
>>>
```

python3 中 execfile 函数已经取消。

```
#exec的第一个参数必须是python代码，一个字符串，或者一个打开的文件里，或者一个编译过的代码对象
#exec执行时，需要解析要执行的代码，这样消耗很大，可以用compile函数先把字符串编译为一个代码对象，然后再执行
code="x='sth'"
x='nth'
exec(code)
print(x)
#eval函数不能执行语句，而执行表达式
z=eval("'xo'*10")
#z=eval(”x=3”)#错误
print(z)
```

2.1.4 sys 模块

```
import sys
>>> dir(sys)

重点成员包括:
'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'set_asyncgen_hooks', '
    set_coroutine_wrapper', 'setcheckinterval', 'setprofile', 'setrecursionlimit', 'setswitchinterval', '
    settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'version', 'version_info'
```

常用属性 argv 包含命令行输入的参数列表 sys.stdout=open('a.log','w') 把输出重定向到一个文件 a.log 中。

path 属性给出 python 的包路径。如果要添加自定义的包路径，可以把包路径放到环境变量 pythonpath 中，该环境变量会自动添加到 sys.path 属性中。另一种方便的方法是，在编程中指定我们的 module 路径 sys.path 中。

2.1.5 re 模块

正则表达式

2.1.6 os 模块

通用的操作系统接口，定义了程序要处理的文件，进场，用户以及线程这些东西。getcwd，返回当前目录的路径字符串 listdir(path)，返回指定目录的所有文件名的列表 chown(path，uid，gid)，改变指定文件的拥有者 id 和组 id chmod(path，mode)，改变指定文件的许可权限为 mode rename(src，dest)，把 src 的文件名改为 dest remove(path) 或 unlink(path)，删除指定的文件 rmdir(path)，删除目录 mkdir(path[,mode])，创建名为 path，权限为 mode 的目录 system(cmd)，在一个 shell 里面执行 shell 命令。symlink(src，dest)，创建从 src 到 dest 的软链接 link(src，dest)，创建从 src 到 dest 的硬链接

os 的重要属性 name，定义当前操作系统的版本 error，定义 os 模块内引入异常使用的类，当异常引发时，该异常带有两个信息，一个是错误号，第二个是解释它的字符串信息。environ，包含当前的环境变量

os 定义的与目录操作相关的字符串 curdir，表示当前目录的字符串 pardir，表示当前目录的父目录的字符串 sep，表示路径名的分隔符 alstsep，另一个分隔符 pathsep，不同路径之间的分隔符

os.path 模块的最常用函数 split(path)，把路径分为两个部分，一是路径，二是文件名 join(path,...)，把各个部分组成一个路径 exists(path)，如果 path 存在返回真 expanduser(path)，用可选的用户名扩展变量，windows 下无效 expandvars(path)，返回对应 path 的环境变量的值 isfile，isdir，islink，ismount，如果 path 是文件，目录，链接，或安装点则返回真 normpath，对路径规范化，去掉多余的分隔符 samefile(p，q)，如果 p，q 引用的是同一个文件返回真 walk(p，visit，arg)，对以 p 开始的目录树里的每个目录调用函数 visit，参数为 arg，dirname，names。其中参数 dirname 是被访问的目录，参数 names 是目录里的文件列表。

2.1.7 subprocess 模块

subprocess 模块用以代替 os.system，os.spawn 模块。允许产生大量新的进程，连接输入/输出/错误管道，并获取返回它们的返回码。(allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.)

2.1.8 shutil 模块

拷贝文件和目录

在 dos 上拷贝一个文件也就是二进制模式打开文件，读入所有数据，再用二进制写模式打开另一个文件，把数据全写到第二个文件中。

在 nnix 和 windows 上这样的拷贝无法拷贝文件相关的 stat 位 (权限，修改时间等)。换句话说拷贝不是那么简单。通常可以使用一个简单的函数 copyfile，术语 shutil 模块。该模块还有一些函数:

copyfile(src，dest)，拷贝 src 文件到 dest 文件 (直接的二进制拷贝) copymod(src，dest)，把 src 的模式信息 (权限许可) 拷贝到 dest 文件 copystat(src，dest)，把 src 的所有状态信息拷贝到 dest 文件 copy(src，dest)，把 src 的数据和状态信息拷贝到 dest，不包括 mac 机上的资源派生 copy2(src，dest)，把 src 的数据和状态信息拷贝到 dest，不包括 mac 机上的资源派生 copytree(src，dest，symlinks=0)，递归的调用 copy2 函数拷贝一个目录。symlinks 标志指示是拷贝链接还是被链接的文件。rmtree(path，ignore_errors=0，onerror=none)，递归的删除 path 目录

2.1.9 cgi 模块

公共网关接口

2.1.10 urllib 和 urlparse 模块

操作 url

特殊的 internet 协议的支持模块 httplib，ftplib，gopherlib，poplib，imaplib，nntplib，smtplib，socketserver，simplehttpserver，cgihttpserver

处理 internet 文件模块: sgmllib，htmllib，xmllib，formatter，rfc822，minetools，binhex，uu，binascii，xdrllib，mimetypes，base64，quopri，mailbox，mimify

2.1.11 struct 模块

处理二进制数据

2.1.12 调试，时间，优化模块

调试器模块 pdb，time 提供很多时间操作函数，profile 模块分析程序，看是否能优化。

2.1.13 copy 模块

浅拷贝 ([:]) 和深拷贝的差别在于，深拷贝完全复制原对象形成了一个新的对象。而浅拷贝，如果拷贝的对象是不变的对象，那么会产生一个新的对象。但如果拷贝的对象是可变的那么拷贝只是产生了对原对象的一个新的引用。

A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. A deep copy constructs a new

compound object and then, recursively, inserts copies into it of the objects found in the original.

2.2 一些应用

2.2.1 排序

参考:sorting how to list.sort() 针对列表而 sorted 可以针对其它对象。可以指定排序的参数，用 key 指定。

在 sorting how to 中还有更复杂的例子，即对类的对象进行排序。

```
>>> sorted([5, 2, 3, 1, 4])

[1, 2, 3, 4, 5]
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})

[1, 2, 3, 4, 5]
>>> sorted("This␣is␣a␣test␣string␣from␣Andrew".split(), key=str.lower)

['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
>>> sorted(student_tuples, key=lambda student: student[2])
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

在列表中随机选择，用 random 模块的的 choice 函数。

2.2.2 新的数据结构

定义一个简单的栈:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

class stack:
    def ___init___(self,data):
        self._data=list(data)
    def push(self,item):
        self._data.append(item)
    def pop(self):
        item=self._data[-1]
        del self._data[-1]
        return item
    def show(self):
```

```
14          print(self._data)
15
16  thingstodo=stack(['write␣to␣mom','write␣to␣dad','write␣to␣bob'])
17  thingstodo.show()
18  print(thingstodo.push('write␣to␣tom'))
19  thingstodo.show()
20  print(thingstodo.pop())
21  thingstodo.show()
```

2.2.3 文件操作

脚本语言的设计目标之一是帮助人们快速而简单地重复工作。web 管理员，系统管理员和程序员经常要做的一件事是，从一个文件集合中选出一个子集，对这个子集做某种操作，并把结果写到一个或一组输出文件中。python 中有一些特定的工具比如 sed 和 awk。

解析一个包含文本的输入文件时,sys 模块非常有用。其中三个文件对象,sys.stdin,sys.stdout,sys.stderr，分别是标准输入，输出，错误。它们与命令行工具有关，print 语句使用标准输出。它是一个文件对象，具有写模式打开的文件对象的所有输出方法如:write 和 writelines。stdin 的输入方法包括:read，readline，readlines。

stringIO 模块可以把字符串封装成一个文件对象。

交互解释器的大部分功能有 cmd 模块的 cmd 类提供

文件操作的内容参考:

python 入门经典 (p289-305)

Chapter 3

python 语法

参考: 像计算机科学家那样思考-理解 python 编程，python 核心编程

3.1 基础知识

3.1.1 程序是什么

程序是一连串具体说明如何计算的指令。这种计算可能是数学的，像是找到方程组的解或是多项式的根，也可能是一种象征性的计算，就像在文件中搜寻并取代文字，或（说来也奇怪）编译一个程序。不同程序语言的详细情况看起来都不一样，但有一些基本的指令，几乎在每种程序语言中都可以发现：输入：从键盘、档案或是其它装置取得数据；输出：在屏幕上显示数据，或着是将数据传送到档案或是其它装置；数学：执行基本的数学运算，如加法和乘法；条件执行：检查特定条件，并执行适当的陈述序列；重复：反复执行某些动作，通常会有些变化。不管你相信与否，就是这么多了。你曾使用过的每个程序，不论有多复杂，都由或多或少类似的指令组成。因此，我们可以把程序设计当成是一种拆解的过程，将大型、复杂的任务，逐步分离成愈来愈小的子任务，直到这些子任务简单到能使用这些基本指令执行为止。

3.1.2 三类错误

程序里有三类可能发生的错误：语法 (句型) 错误、执行错误以及语意错误

语法错误（Syntax errors）

Python 只能够执行语法正确的程序，否则程序就会执行失败，并传回错误讯息。语法就是指程序的结构，以及结构的规则。

执行错误（Runtime errors）

第二种类型的错误叫做执行错误，会这么命名是因为这种错误直到执行的时候才会出现。这种错误也称为异常，因为它们通常表示某种异常（而且不好的）事情发生。

语意错误（Semantic errors）

第三种类型错误叫做语意错误。如果有语意错误在你的程序里，程序仍会顺利的执行，因此计算机不会产生任何的错误讯息，但是程序不会做正确的事情。程序还是会执行另一些事情，特别是你叫程序执行的事。问题是你写的程序并非是你想要写的程序。程序的意义（它的语意）是错的。判定语意错误可能是困难的，因为需要你检视程序的输出，并尝试找出程序正在执行的事，以回溯你的工作。

3.1.3 形式语言及自然语言

自然语言是人们所讲的语言，如英语、西班牙语和法语。它们并非是由人所设计的（虽然人们尝试将某种规则套用在它们上面），而是自然演变而成。

形式语言是人们为特定应用所设计的语言。举例来说，数学家所使用的标记法就是一种形式语言，这种语言特别适合表示数字与符号间的关系。化学家也使用一种形式语言表现分子的化学结构。

最重要的是：程序语言是设计来呈现计算的形式语言。

形式语言对于语法有严格的规则。语法规则分为两种，分别属于标记与结构。标记是程序语言的基本组件，第二种语法规则属于陈述的结构—也就是说，标记的排列方法。

3.2 python 语法

3.2.1 常量类型

使用 type() 可以查看常量的类型

```
>>> type(1)
<class 'int'>
>>> type('str')
<class 'str'>
>>> type(3.2)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> def func():
        pass

>>> type(func)
<class 'function'>
```

3.2.2 变量与保留关键字

程序设计师通常会为变量选择一个有意义的名称，它们记录了该变量的用途。变量名称可以为任意长度，也可以同时包含字母与数字，但必须以字母开头。虽然也可以使用大写字

母，但通常我们不如此做。如果你同时使用大小写字母，请记住大小写有分别。如 Bruce 与 bruce 是不同的变数。下滑线符号可以出现在变量名称中。它通常用于多字的的名称中，例如 my_name 或 price_of_tea_in_china。

变量的赋值

```
#!D:\MinGW\python\python.exe
#!/usr/bin/env python
#_*_utf-8_*_


print #空输出空一行
print '-----------------------------------'
print 'python基本原则:'
print '-----------------------------------'

print 'hello world!'
print "hello python!"
i=0
while i<256: #冒号表示代码块
    if i!=0 and i%8==0:
        print i,chr(i)
    i=i+1
print "please put in any key to continue:" #print输出
#a=raw_input() #raw_input()输入
a=1
A="A" #等号是赋值，其本质是变量指向在某一内存空间中的值。
b=A
A="B" #变量指向变量，则直接指向值。
print 'you put in key:', a, b #大小写敏感，输出变量值
a_1_b="变量名" #变量名可以用字母数字下划线表示，不以数字开头
```

Python 与大多数其它语言一样有局部变量和全局变量之分，但是它没有明显的变量声明。变量通过首次赋值产生，当超出作用范围时自动消亡。

变量的赋值是一条被分成了多行的命令，用反斜线（“\”）作为续行符。当一条命令用续行符（“\”）分割成多行时，后续的行可以以任何方式缩进，此时 Python 通常的严格的缩进规则无需遵守。如果您的 Python IDE 自由对后续行进行了缩进，您应该把它当成是缺省处理，除非您有特别的原因不这么做。

Python 不允许您引用一个未被赋值的变量，试图这样做会引发一个异常。

给多个变量赋值:

```
>>> v=("a","b","c")
>>> v
('a', 'b', 'c')
>>> (x,y,z)=v
>>> x
'a'
>>> y
```

```
'b'
>>> z
'c'
>>> [l,m,n]=v
>>> l
'a'
>>> m
'b'
>>> n
'c'
>>>
```

在 C 语言中，可以使用 enum 类型，手工列出每个常量和其所对应的值，当值是连续的时候这一过程让人感到特别繁琐。而在 Python 中，您可以使用内置的 range 函数和多变量赋值的方法来快速进行赋值。

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7)
>>> MONDAY
0
>>>
```

内置的 range 函数返回一个元素为整数的 list。这个函数的简化调用形式是接收一个上限值，然后返回一个初始值从 0 开始的 list，它依次递增，直到但不包含上限值。(如果您愿意，您可以传入其它的参数来指定一个非 0 的初始值和非 1 的步长。也可以使用 print range.___doc___ 来了解更多的细节。

关键词用来定义程序语言的规则与结构，不能当成变量名称。Python 有 31 个关键词：and del from not while as elif global or with assert else if pass yield break except import print class exec in raise continue finally is return def for lambda try

3.2.3 字符编码

```
print ord('a'), chr(65) #asc码转换
u'中' #中的unicode编码
u'中'.encode('utf-8') #中的utf-8编码
'\xe4\xb8\xad'.decode('utf-8') #utf-8编码对应的unicode码
print u'中', u'中'.encode('utf-8'), '\xe4\xb8\xad'.decode('utf-8')
#输出时,都是字符本身,而不是编码,除非在命令行中使用上述命令可以显示编码
```

3.2.4 计算表达式

3.2.5 运算符和规则

```
print 10/3, 10/3.0, 10%3 #整数除法永远是整数,若有一个浮点数则会变成浮点,求余数用%
注意:
/和//的差异
```

3.2.6 输入与输出

输入提示

3.2.7 json 格式文件读写

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

"""
测试json格式
"""
import json

#编码成json格式，使用dumps函数
i=['python',[1,2,3],{'name':'xiaoming'},("abc",123)]
encoded_json=json.dumps(i)
print(repr(i))
print(encoded_json)

#从json格式解码，使用loads函数
decoded_json=json.loads(encoded_json)
print(type(decoded_json))
print(decoded_json)


#将对象写入文件，使用dump函数
f = open("testjsonfmt.dat","w")
file_json=json.dump(i,f)
f.close()

#从文件读取json格式对象，使用load函数
f=open("datatopython.dat","r")
data=json.load(f)
f.close()
```

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：

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='Bob', age=20, score=88)
>>> json.dumps(d)
```

JSON 类型	Python 类型
{}	dict
[ ]	list
”string”	’str’ 或 u’unicode’
1234.56	int 或 float
true/false	True/False
null	None

表 3.1: json 类型

```
'{"age": 20, "score": 88, "name": "Bob"}'
```

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：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{u'age': 20, u'score': 88, u'name': u'Bob'}
```

有一点需要注意，就是反序列化得到的所有字符串对象默认都是 unicode 而不是 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 或 unicode 与 JSON 的字符串之间转换。

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('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个 TypeError：

```
Traceback (most recent call last):
  ...
TypeError: <__main__.Student object at 0x10aabef50> is not JSON serializable
```

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：https://docs.python.org/2/library/json.html#json.dumps 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：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }

print(json.dumps(s, default=student2dict))
```

这样，Student 实例首先被 student2dict() 函数转换成 dict，然后再被顺利序列化为 JSON。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))
```

运行结果如下：

```
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的 Student 实例对象。小结 Python 语言特定的序列化模块是 pickle，但如果要把序列化搞得更通用、更符合 Web 标准，就可以使用 json 模块。json 模块的 dumps() 和 loads() 函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。

3.2.8 注释

3.2.9 逻辑表达式与逻辑运算

and 返回第一个假值或最后一个真值

or 返回第一个真值或最后一个假值

一个好的例子见3.2.10节。

3.2.10 条件控制

if elif else

单句的条件有两种方式，一种是用逻辑返回实现，另一种是用单句条件控制比如:

```
>>> f=lambda x,y: (x<y and [x] or [y])[0]
>>> f(4,8)
4
>>> g=lambda x,y:x if x<y else y
>>> g(4,6)
4
```

if elif 类似于 c 中的 switch 语句，fortran 中 select case 语句。

```
print #空输出空一行
print '----------------------------------'
print '结构:条件选择和循环'
print '----------------------------------'
age=18
if age<18: #if条件选择,elif,else共用
    print '青少年'
elif age<45:
    print '青年'
elif age<60:
    print '中年'
else:
    print '老年'
```

3.2.11 循环

break 离开循环体

continue 与 c++ 等语言一致

else 是循环运行结束时运行的语句，但 break 跳出的例外

注意:range 是到 <stop 的数为止，比如 range(10)，那么范围是 (0 到 9) range(1,1)，那么范围为空，而 range(1,2) 范围才存在。

```
for vara in ['a','b','c', 45, False]: #使用for in循环
    print vara
```

```
3
4  suma=0
5  for vara in range(101):
6      suma=suma+vara
7  print suma
8
9  suma=0 #使用while循环
10 n=100
11 vara=1
12 while vara<=n:
13     suma=suma+vara
14     vara=vara+1
15 print suma
```

For in 循环

range 生成一个整数的 list，通过它来控制循环。for 循环不仅仅用于简单计数。它们可以遍历任何类型的东西，比如 dict 的例子。

```
1  for i in range(5):
2      print (i)
3
4  li = ['a', 'b', 'c', 'd', 'e']
5  for i in range(len(li)):
6      print (li[i])
7
8
9  import os
10 for k, v in os.environ.items():
11     print ("%s=%s" % (k, v))
12
13 print ("\n".join(["%s=%s" % (k, v) for k, v in os.environ.items()]))
```

利用 for in 实现抽象迭代: 的确很自由

```
1  print #空输出空一行
2  print '----------------------------------'
3  print '利用for in实现抽象迭代:的确很自由'
4  print '----------------------------------'
5  #for in除了可以在list,tuple等数组内遍历,也可以在dict,set,字符串内进行遍历
6  lista=('abc',123,1.5,True)
7  tuplea=('abc',123,1.5,True,[3, 4])
8  dicta={'zhangsan':95,'lisi':80,'wangwu':55}
9  seta=set([7,8,9])
10
11 i=0
12 for var in lista:
13     i=i+1
14     print 'in lista:',i,var
15
16 i=0
```

```
for var in tuplea:
    i=i+1
    print 'in tuplea:',i,var

i=0
for var in dicta:
    i=i+1
    print 'in dicta:',i,var
i=0
for var in dicta.itervalues():
    i=i+1
    print 'in dicta:',i,var
i=0
for var in dicta.iteritems():
    i=i+1
    print 'in dicta:',i,var
i=0
for var in dicta.items():
    i=i+1
    print 'in dicta:',i,var

i=0
for var in seta:
    i=i+1
    print 'in seta:',i,var

i=0
for var in 'abcdefg':
    i=i+1
    print 'in string:',i,var

#使用isinstance函数,用Iterable属性,判断是否可以迭代遍历
#注意来自模块collections
from collections import Iterable
print isinstance('abc',Iterable) # str是否可迭代
print isinstance([1,2,3], Iterable) # list是否可迭代
print isinstance(123, Iterable) # 整数是否可迭代

#利用enumerate函数可以遍历索引和元素本身
for j,var in enumerate(dicta.iteritems()):
    print 'in dicta:',j,var

#两个变量的元素的遍历
for x,y in [(1,2),(3,4),(5,6)]:
    print 'in list:',x,y
for (x,y) in [(1,2),(3,4),(5,6)]:
    print 'in list:',(x,y)
#注意:两个变量的元素可以用list,tuple,set,dict给出都可以。
for (x,y) in [[1,2],(3,4),([5,6]),{"key1":63,"key2":96}]:
    print 'in list:',(x,y)
```

3.2.12 判断表达式

布尔值，它的值或者为 True 或者为 False。请注意第一个字母是大写的也可以使用其他方式替代: 0 为 false; 其它所有数值皆为 true。空串 (””) 为 false; 其它所有字符串皆为 true。空 list ([]) 为 false; 其它所有 list 皆为 true。空 tuple (()) 为 false; 其它所有 tuple 皆为 true。空 dictionary () 为 false; 其它所有 dictionary 皆为 true。

And 和 or

在 Python 中，and 和 or 执行布尔逻辑演算，如你所期待的一样。但是它们并不返回布尔值，而是返回它们实际进行比较的值之一。And 如果布尔环境中的某个值为假，则 and 返回第一个假值。在这个例子中，” 是第一个假值。所有值都为真，所以 and 返回最后一个真值，’c’。Or 如果有一个值为真，or 立刻返回该值。如果所有的值都为假，or 返回最后一个假值。or 演算” 的值为假，然后演算 [] 的值为假，依次演算 的值为假，最终返回 。如果你是一名 C 语言黑客，肯定很熟悉 bool ? a : b 表达式，如果 bool 为真，表达式演算值为 a，否则为 b。基于 Python 中 and 和 or 的工作方式，你可以完成相同的事情。

3.2.13 类型转换

3.2.14 图形接口 (API)

3.6.2 中没有 gasp 模块

3.2.15 逸出序列

字符串\t 代表一个 tab 字符。在\t 中的反斜线指出这是逸出序列的开始。跳脱序列用来表示不可见的字符，像是 tab 和新行。\n 序列就表示一个新行。跳脱序列能够出现在字符串的任何地方；在这个例子中，tab 跳脱序列是这个字符串里唯一的内容。

3.2.16 算法

算法的一个特征就是不需要智力完成。算法是按照简单法则一步接一步的机械式步骤。

牛顿求根法:

```
1  #!/usr/bin/env python3
2  #_*_coding: utf-8 _*_
3
4  def sqrta(s):
5      """
6      求近似根算法:
7      r(n+1)=[r(n)+s/r(n)]/2
8      r(0)=s/2
9      :param n:
10     :return:
```

```
    """
    approx = s/2.0
    better = (approx + s/approx)/2.0
    while abs(better-approx)> 1e-14:
        approx = better
        better = (approx + s/approx)/2.0
    return approx

print (sqrta(25))
print (sqrta(9))
print (sqrta(5))
print (sqrta(2))
```

相对于为单一问题撰写一个特定解决方法，当你为一种类别的问题撰写一般化的解决方式时，你就写出了一个算法（algorithm）。我们在之前已经提过这个词，不过并没有特意定义它。它并不容易定义，所以我们会尝试一些方法。

一样的道理，你所学过加法的进位、减法的借位以及长除法都是算法。算法的一个特征就是不需要智力完成。算法是按照简单法则一步接一步的机械式步骤。

依我们的看法，人类在学校花太多时间练习执行实际上不需智力的算法，是很丢脸的。另一方面来说，设计算法的过程是有趣而且考验智力的，而其核心部份就是我们所说的程序设计。

有一些人们做起来很自然、毫无困难或知觉的事是最难用算法表达的。理解自然语言就是个好例子。我们所有人都说话，但直到现在没有人可以解释我们如何说话，至少不是用算法的型式。

3.2.17 字符串

由小片段组成的类型叫做复合数据类型。依据我们所做的事，我们可能会想要将复合数据类型视为单一的整体，或者我们也可能想要存取它的一部分。这种模棱两可的情况是有用的。

中括号里的表达式叫做索引。索引指定一个有序集合的成员，在这个例子是字符串中字符的集合。索引指出那一个成员是你想要的，也就是就是成员的名称。它可以是任何的整数表达式。

```
fruit = "banana"
letter = fruit[1]
print(letter)
```

len 函数回传字符串里字符的数量

字符串是不可变的，这个意思是说你不能改变已存在的字符串内容。最好的做法是建立一个新的字符串。

in 运算子测试一个字符串是否为另一个字符串的子字符串

字符串有非常多的操作函数，比如:startwith，swapcase 等等

字符串格式化

字符串格式化不只是连接。它甚至不仅仅是格式化。它也是强制类型转换。如同 printf 在 C 中的作用，Python 中的字符串格式化是一把瑞士军刀。它有丰富的选项，不同的格式化格式符和可选的修正符可用于不同的数据类型。

数值的格式化

%f 格式符选项对应一个十进制浮点数，不指定精度时打印 6 位小数。使用包含“.2”精度修正符的%f 格式符选项将只打印 2 位小数。您甚至可以混合使用各种修正符。添加 + 修正符用于在数值之前显示一个正号或负号。注意“.2”精度修正符仍旧在它原来的位置，用于只打印 2 位小数。

```
>>> print ("price=%f"% 50.4625)
price=50.462500
>>> print ("price=%.2f"% 50.4625)
price=50.46
>>> print ("price=%+.5f"% 50.4625)
price=+50.46250
>>>
```

不同数据类型的格式化输出

```
print #空输出空一行
print '----------------------------------'
print 'python数据类型和格式化输出:'
print '----------------------------------'

a=True #布尔数,True/False
b=256 #整数
c=1.21e3 #浮点数
d="string字符串" #字符串
e='单引号\'双引号"都可用于表示字符串\n换行\t制表符\\'
f=r'单引号\'双引号"都可用于表示字符串\n换行\t制表符\\'␣#r'...'表示不转义
g='''多行
多行
多行'''␣␣#'''...'''可以表示多行
h=None␣␣␣#空值用None表示
print␣a,␣b,␣c,␣d,␣e,␣f,␣g,␣h
print␣'格式化输出布尔数%s,%s,'%(a,␣False)
print␣'格式化输出整数%d,%4d,%05d'%(b,␣b,␣b)
print␣'格式化输出浮点数%f,%4.0f,%5.4f'%(c,␣c,␣c)
print␣'格式化输出浮点数%e,%4.0e,%5.4e'%(c,␣c,␣c)
print␣'格式化输出浮点数%E,%4.0E,%5.4E'%(c,␣c,␣c)
print␣'格式化输出浮点数%g,%4.0g,%5.4g'%(c,␣c,␣c)
print␣'格式化输出浮点数%G,%4.0G,%5.4G'%(c,␣c,␣c)
print␣'格式化输出字符串和字符%s,%s,%c'%(d,␣'c',␣'c')
```

列表中字符串的链接与分割

为了将任意包含字符串的 list 连接成单个字符串，可以使用字符串对象的 join 方法。join

只能用于元素是字符串的 list；它不进行任何的强制类型转换。连接一个存在一个或多个非字符串元素的 list 将引发一个异常。

```
>>> ";".join(["b","c","a"])
'b;c;a'
>>> 'b;c;a'.split(";")
['b', 'c', 'a']
>>>
```

没有参数的 split 函数。不带参数它按空白进行分割。

3.2.18 元组与不可变性

Tuple 是不可变的 list。一旦创建了一个 tuple，就不能以任何方式改变它。定义 tuple 与定义 list 的方式相同，但整个元素集是用小括号包围的，而不是方括号。Tuple 的元素与 list 一样按定义的次序进行排序。Tuples 的索引与 list 一样从 0 开始，所以一个非空 tuple 的第一个元素总是 t[0]。负数索引与 list 一样从 tuple 的尾部开始计数。与 list 一样分片 (slice) 也可以使用。注意当分割一个 list 时，会得到一个新的 list ；当分割一个 tuple 时，会得到一个新的 tuple。Tuple 没有方法您可以使用 in 来查看一个元素是否存在于 tuple 中。dictionary keys 可以是字符串，整数和“其它几种类型”吗？Tuples 就是这些类型之一。Tuples 可以在 dictionary 中被用做 key，但是 list 不行。实际上，事情要比这更复杂。Dictionary key 必须是不可变的。Tuple 本身是不可改变的，但是如果您有一个 list 的 tuple，那就认为是可变的了，用做 dictionary key 就是不安全的。只有字符串、整数或其它对 dictionary 安全的 tuple 才可以用作 dictionary key。Tuple 可以转换成 list，反之亦然。内置的 tuple 函数接收一个 list，并返回一个有着相同元素的 tuple。而 list 函数接收一个 tuple 返回一个 list。从效果上看，tuple 冻结一个 list，而 list 解冻一个 tuple。

```
a=("str1")
print(type(a))
a=("str1",)
print(type(a))

结果:
<class 'str'>
<class 'tuple'>
```

```
tuplea=(1,) #一个元素的tuple在元素后加逗号消除歧义
print tuplea
tuplea=('abc',123,1.5, True)
print tuplea
tuplea=('abc',123,1.5,True,[3, 4])
print tuplea
tuplea[4][1]='a' #无法修改tuple的元素,但可以对tuple中嵌套的list的元素进行重新赋值
print tuplea
```

keys, values 和 items 函数 Dictionary 的 keys 方法返回一个包含所有键的 list。这个 list 没按 dictionary 定义的顺序输出 (记住，元素在 dictionary 中是无序的)，但它是一个 list。values 方法返回一个包含所有值的 list。它同 keys 方法返回的 list 输出顺序相同，所以对于所有的 n，params.values()[n] == params[params.keys()[n]] 。items 方法返回一个由形如 (key，value) 组成的 tuple 的 list。这个 list 包括 dictionary 中所有的数据。

3.2.19 列表、数组

List 是一个用方括号包括起来的有序元素的集合。List 可以作为以 0 下标开始的数组。任何一个非空 list 的第一个元素总是 li[0]。包含 5 个元素 list 的最后一个元素是 li[4]，因为列表总是从 0 开始。负数索引从 list 的尾部开始向前计数来存取元素。任何一个非空的 list 最后一个元素总是 li[-1]。可以这样理解：li[-n] == li[len(li) - n]。所以在 5 个数的 list 里，li[-3] == li[5 - 3] == li[2]。切片用的也是索引，但第二个数表示取到这个索引的前一个数为止。比如:li[1:3] 为 li[1],li[2] append 向 list 的末尾追加单个元素。insert 将单个元素插入到 list 中。数值参数是插入点的索引。请注意，list 中的元素不必唯一，现在有两个独立的元素具有'new' 这个值，li[2] 和 li[6]。extend 用来连接 list。请注意不要使用多个参数来调用 extend，要使用一个 list 参数进行调用。index 在 list 中查找一个值的首次出现并返回索引值。index 在 list 中查找一个值的首次出现。这里'new' 在 list 中出现了两次，在 li[2] 和 li[6]，但 index 只返回第一个索引，2。如果在 list 中没有找到值，Python 会引发一个异常。这一点与大部分的语言截然不同，大部分语言会返回某个无效索引。尽管这种处理可能令人讨厌，但它仍然是件好事，因为它说明您的程序会由于源代码的问题而崩溃，好于在后面当您使用无效索引而引起崩溃。要测试一个值是否在 list 内，使用 in。如果值存在，它返回 True，否则返为 False

remove 从 list 中删除一个值的首次出现。remove 仅仅删除一个值的首次出现。在这里，'new' 在 list 中出现了两次，但 li.remove("new") 只删除了'new' 的首次出现。如果在 list 中没有找到值，Python 会引发一个异常来响应 index 方法。pop 是一个有趣的东西。它会做两件事：删除 list 的最后一个元素，然后返回删除元素的值。请注意，这与 li[-1] 不同，后者返回一个值但不改变 list 本身。也不同于 li.remove(value)，后者改变 list 但并不返回值。

Lists 也可以用 + 运算符连接起来。list = list + otherlist 相当于 list.extend(otherlist)。但 + 运算符把一个新 (连接后) 的 list 作为值返回，而 extend 只修改存在的 list。也就是说，对于大型 list 来说，extend 的执行速度要快一些。Python 支持 += 运算符。li += ['two'] 等同于 li.extend(['two'])。+= 运算符可用于 list、字符串和整数，并且它也可以被重载用于用户自定义的类中。* 运算符可以作为一个重复器作用于 list。li = [1, 2] * 3 等同于 li = [1, 2] + [1, 2] + [1, 2]，即将三个 list 连接成一个。

```
print #空输出空一行
print '----------------------------------'
print '数组:list,tuple,dict,set:'
print '----------------------------------'
```

```
5
6  lista=['abc',123,1.5, True]
7  print lista
8  print lista[0], lista[3] #若list中共n个元素,取值位置范围是0~n-1
9  print lista[-4], lista[-1] #或者从-n~-1
10 print 'list长度:', len(lista)
11 lista.append('appendedstr') #在后面添加
12 print lista
13 lista.insert(4,'insertedstr') #在某位置前面插入,位置是从0~开始计算
14 print lista
15 lista.insert(0,'firststr') #在某位置前面插入,位置是从0~开始计算
16 print lista
17 lista.pop() #删除最后一个元素
18 print lista
19 lista.pop(0) #删除某一位置的元素
20 print lista
21 lista[4]='modifiedstr' #对某一位置元素重新赋值
22 print lista
23 lista.insert(4,['nested0','nested1']) #list可以嵌套
24 print lista
25 print lista[4][0], lista[4][1] #嵌套中的list的元素的取值
26 print lista[4][-2], lista[4][-1] #嵌套中的list的元素的取值
```

n 个 0 的列表:

```
a=[0]*n
```

尽管 python 中没有数组这一概念，但本质上列表就是数组。比如:

```
mat = [
 [0, 0, 2,0],
 [1, 1, 1,0],
 [0, 1, 0,0]]

def GetMatVal(i,j):
   return mat[i][j]


print(GetMatVal(0,0))
print(mat[0][0])
print(len(mat))
print(mat[2][1])
```

其中，mat 就是一个 3 行 4 列的数组，引用的时候行在前，列在后。也就是用 [][] 解析的时候，第一个参数是解析列表中第一层的数据，第二参数解析的是该数据中的内部数据。比如 [0][2] 第一个 0 获取 [0, 0, 0,0]，第二个获取其中的第 3 个数 2。注意: 列表的索引是从 0 开始的，因此最大索引小于其长度，即等于长度-1。

列表解析方法

```
savedidxlista=[1,2,3,-1,-2]
fa=lambda idx : len(codea)+idx if idx<0 else idx #lambda 函数将负的索引值换成正的
idxlista=[fa(x) for x in savedidxlista]
```

```
a=[[1,2],[2,3],[3,4]]
for x,y in a:
    print(x,y)

b=[1,2,3,4]
c=[[b[i],b[i+1]] for i in range(len(b)-1)]
print(c)
```

数组的切片取值

```
print #空输出空一行
print '----------------------------------'
print '数组的切片取值:'
print '----------------------------------'
#类似与fortran中的数组取值方法,并不奇特
lista=('abc',123,1.5,True)
tuplea=('abc',123,1.5,True,[3, 4])
print lista[1:3]
print lista[::2]
print tuplea[1:3]
print tuplea[::2]
```

列表生成

```
print #空输出空一行
print '----------------------------------'
print '列表生成'
print '----------------------------------'
print range(1, 11) #range函数

L = [] #list用for循环添加
for x in range(1, 11):
    L.append(x * x)
print L

#直接利用for in生成list,可以添加约束if
print [x*x for x in range(1, 11) if x%2 == 0]

#字符串也可以类似处理,并进行两层循环
print [m + n for m in 'ABC' for n in 'XYZ']
print [m*n for m in range(1, 6) for n in range(6, 11)]

```

```
19 import os #导入os模块，模块的概念后面讲到
20 lista=[d for d in os.listdir('.')] #os.listdir可以列出文件和目录
21 for i in range(len(lista)):
22     print i, lista[i]
23
24 d = {'x': 'A', 'y': 'B', 'z': 'C' } #两个变量同时用
25 print [k + '=' + v for k, v in d.iteritems()]
26
27 L = ['Hello', 'World', 'IBM', 'Apple']
28 print [s.lower() for s in L] #小写字母
29 print [s.upper() for s in L] #大写字母
30
31 x = 'abc'
32 y = 123
33 print isinstance(x, str) #利用isinstance函数和str属性判断是否为字符串
34 print isinstance(y, str)
35
36 #在Python中，这种一边循环一边计算的机制，称为生成器（Generator）
37 #创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator
38 L = [xa * xa for xa in range(10)]
39 g = (x * x for x in range(10))
40 print L
41 print g
42 print g.next() #用next函数可以逐个打印
43 for n in g: #最好是用for来打印
44     print n
45
46 def fab(max):
47     n, a, b = 0, 0, 1
48     while n < max:
49         yield b #用yield代替print就可以将直接产生list的函数变成为generator
50         a, b = b, a + b
51         n = n + 1
52 print fab(6)
53 for n in fab(6): #最好是用for来打印
54     print n
55
56 #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。
57 #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
58 #回的yield语句处继续执行。
```

列表映射和过滤

映射是其对 list 的解析，它提供一种紧凑的方法，可以通过对 list 中的每个元素应用一个函数，从而将一个 list 映射为另一个 list。列表过滤，过滤列表语法：[mapping-expression for element in source-list if filter-expression] 前三部分与映射都是相同的；最后一部分，以 if 开头的是过滤器表达式。过滤器表达式可以是返回值为真或者假的任何表达式 (在 Python 中

是几乎任何东西)。任何经过滤器表达式演算值为真的元素都可以包含在映射中。其它的元素都将忽略，它们不会进入映射表达式，更不会包含在输出列表中。

Count 方法

count 是一个列表方法，返回某个值在列表中出现的次数。

3.2.20 字典

```
#在任何时候都可以加入新的 key−value 对。这种语法同修改存在的值是一样的。
#Di cti onary 的 key 是大小写敏感的
#Dictionary 不只是用于存储字符串。Dictionary 的值可以是任意数据类型， 包括字符串、整数、对象，甚至其它的 dictionary。在单个 dictionary 里， dictionary 的值并不需要全都是同一数据类型，可以根据需要混用和匹配。
#Dictionary 的 key 要严格多了，但是它们可以是字符串、整数或几种其它 的类型 (后面还会谈到这一点)。也可以在一个 dictionary 中混用和匹配 key 的数据类型。
dicta={'zhangsan':95,'lisi':80,'wangwu':55} #dict快速通过key查找值,用的是hash算法,存储与key顺序无关
print dicta
print dicta['wangwu']
print 'lisi'in dicta
print 'zhaosi'in dicta
print dicta.get('zhangsan')
print dicta.get('zhaosi')
dicta['zhaosi']=61 #可以直接放入key和数据
dicta['lisi']=70 #修改key对应值的方法类似,也是直接改
print dicta
dicta.pop('zhaosi') #删除key用pop命令
print dicta

>>> d={'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
>>> del d[42]
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear()
>>> d
{}
>>>
```

3.2.21 数据集

```
seta=set([1,2,3]) #set类似与数学上的集,不应有重复数据,集可以做并集,交集等处理。
print seta
seta.add(4)
print seta
seta.add(3)
```

```
print seta
setb=set([1,2,6,7])
print setb
print seta&setb #求交集
print seta|setb #求并集

print '可变对象和不可变对象:'
a=['b','a','c']
print a
a.sort()
print a
a='abc'
print a
a.replace('a','A')
print a
print a.replace('a','A')
```

3.2.22 随机数

```
import random
for i in range(10):
    x = random.random()
    print (x)
```

```
i=random.randrange(len(mbs))
```

3.2.23 文件

read(5) 可以指定读取的字符数 readline() 读取一行 readlines() 读取所有行，返回一个列表 open("/usr/share/dict/words","r") open 可以指定路径

```
f = open("test.dat","w")
f.write("Now is the time \t")
f.write("to close the file \n")
f.close()

f = open("test.dat","r")
text = f.read()
print(text)
f.close()
```

再举个例子:

```
spath="baa.dat"
f=open(spath,"w")
f.write("first line \n")
f.writelines("second line 2")
```

```
f.close()
f=open(spath,"r")
for line in f:
    print (line)
f.close()
```

需要注意文件的默认编码是与系统的默认编码相关的，比如中文 win 下的默认编码是 gbk。如果文件是 utf-8 格式的，那么文件打开可以用 codecs 模块的 open，比如:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

"""
测试文件的自动编码格式
"""

#import os
import sys
import codecs


def filedefaulten():#英文字符默认编码为utf-8,只有英文无所谓编码是utf-8还是其它，因为都是一样的
    fpath="filecodea.dat"
    f = open(fpath,"w")
    f.write("Now is the time \t")
    f.write("to close the file \n")
    f.close()
    f = open("testa.dat","r")
    text = f.read()
    print(text)
    f.close()

def filedefaultcn(): #带有中文字符，则默认编码为系统默认编码，win下是gbk
    fpath="filecodeb.dat"
    f = open(fpath,"w")
    f.write("Now is the time \t")
    f.write("to close the file \n")
    f.write("中文 \n")
    f.close()
    f = open(fpath,"r")
    text = f.read()
    print(text)
    f.close()

def filecncodec(): #带有中文字符，利用codec编码为utf-8
    fpath="filecodec.dat"
    f = codecs.open(fpath,"w",encoding="utf-8")
    f.write("Now is the time \t")
    f.write("to close the file \n")
    f.write("中文 \n")
```

```
42      f.close()
43      f = codecs.open(fpath,"r",encoding="utf-8")
44      text = f.read()
45      print(text)
46      f.close()
47
48  def fileencodec(): #只有英文字符，利用codec编码为gbk，只有英文无所谓编码是utf-8还是其它，因为都是一样的
49      fpath="filecoded.dat"
50      f = codecs.open(fpath,"w",encoding="gbk")
51      f.write("Now is the time \t")
52      f.write("to close the file \n")
53      f.close()
54      f = codecs.open(fpath,"r",encoding="gbk")
55      text = f.read()
56      print(text)
57      f.close()
58
59
60
61  def main():
62      print("测试程序!")
63      filedefaulten()
64      filedefaultcn()
65      filecncodec()
66      fileencodec()
67      pass
68
69  if __name__ == "__main__":
70      print('default encoding is:',sys.getdefaultencoding())
71      main()
```

文件中的位置:

seek(pos, whence=0) Set the chunk’s current position. The whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

tell() Return the current position into the chunk.

```
1   #!/usr/bin/env python3
2   #_*_coding: utf-8 _*_
3
4   """
5   测试文件的内的位置移动，文件就是一个字节序列或字节块
6   """
7
8   import os
9
10  #seek的第二个参数，0表示相对于文件头，1表示相对于当前位置，2表示相对于文件尾
```

```
#注意python3中seek只能作用于b模式打开的文件，如果没有b会提示错误:io.UnsupportedOperation: can't do nonzero cur
    -relative seeks
#而python2中则没有这个问题
f=open("testb.dat","wb+")

print('cur pos=',f.tell())
f.write(bytes("abcd\n",encoding="utf8"))
print('cur pos=',f.tell())
f.write("efgh\n".encode())
print('cur pos=',f.tell())

f.seek(-5, 1)
print('cur pos=',f.tell())
print(str(f.readline(),encoding='utf-8'),end='')
print('cur pos=',f.tell())
f.seek(0,0)
print(f.readline().decode(),end='')
print('cur pos=',f.tell())
print(str(f.readline(),encoding='utf-8'))
print('cur pos=',f.tell())

f.write(bytes("中文\n",encoding="utf8"))
print('cur pos=',f.tell())
f.write("字符\n".encode())
print('cur pos=',f.tell())

f.seek(-4,2)
print(str(f.readline(),encoding='utf-8'))
print('cur pos=',f.tell())
f.seek(-11,2)
print(f.readline().decode())
print('cur pos=',f.tell())
f.close()
```

3.2.24 异常和断言

每当一个运作错误发生时，它会建立一个异常 (例外)。通常，这个程序会停止，然后 Python 会印出一个错误讯息。

一个数字除以 0 会产生一个例外存取一个不存在的列表对象也会存取一个不在 dictionary 中的键值打开一个不存在的文件

用 try 和 except 处理异常

举个例子:

```
s=input("input your age:")
if s=="":
    raise Exception("input must not be empty")

```

```
try:
    i=int(s)
except ValueError:
    print("could not convert data to an integer")
except:
    print("unknown exception")
else:#当try没有产生异常时执行
    print("you are %d" % i,"years old")
finally:#总会执行
    print("bye!")
```

assert 用了什么某个条件是真的，如果非真，则引发异常。断言 assert，等于 raise-if-not，即测试一个表达式，如果返回为假，则触发异常，如果不同 try-except 捕捉异常，则会中断并提供 traceback。比如:

```
>>> assert 1==0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    assert 1==0
AssertionError
```

Python 使用 try...except 来处理异常，使用 raise 来引发异常。Java 和 C++ 使用 try...catch 来处理异常，使用 throw 来引发异常。

在程序运行过程中，总会遇到各种各样的错误。有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为 bug，bug 是必须修复的。有的错误是用户输入造成的，比如让用户输入 email 地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。Python 内置了一套异常处理机制，来帮助我们进行错误处理。

错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数 open()，成功时返回文件描述符（就是一个整数），出错时返回-1。用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
```

```
    return r

def bar():
    r = foo()
    if r==(-1):
        print 'Error'
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。所以高级语言通常都内置了一套 try...except...finally... 的错误处理机制，Python 也不例外。try 让我们用一个例子来看看 try 的机制：

```
try:
    print 'try...'
    r = 10 / 0
    print 'result:', r
except ZeroDivisionError, e:
    print 'except:', e
finally:
    print 'finally...'
print 'END'
```

当我们认为某些代码可能会出错时，就可以用 try 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 except 语句块，执行完 except 后，如果有 finally 语句块，则执行 finally 语句块，至此，执行完毕。上面的代码在计算 10 / 0 时会产生一个除法运算错误：

```
try...
except: integer division or modulo by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句 print 'result:', r 不会被执行，except 由于捕获到 ZeroDivisionError，因此被执行。最后，finally 语句被执行。然后，程序继续按照流程往下走。如果把除数 0 改成 2，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以 except 语句块不会被执行，但是 finally 如果有，则一定会被执行（可以没有 finally 语句）。你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的 except 语句块处理。没错，可以有多个 except 来捕获不同类型的错误：

```
try:
    print 'try...'
```

```
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
finally:
    print 'finally...'
print 'END'
```

int() 函数可能会抛出 ValueError，所以我们用一个 except 捕获 ValueError，用另一个 except 捕获 ZeroDivisionError。此外，如果没有错误发生，可以在 except 语句块后面加一个 else，当没有错误发生时，会自动执行 else 语句：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
else:
    print 'no␣error!'
finally:
    print 'finally...'
print 'END'
```

Python 的错误其实也是 class，所有的错误类型都继承自 BaseException，所以在使用 except 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except StandardError, e:
    print 'StandardError'
except ValueError, e:
    print 'ValueError'
```

第二个 except 永远也捕获不到 ValueError，因为 ValueError 是 StandardError 的子类，如果有，也被第一个 except 给捕获了。Python 所有的错误都是从 BaseException 类派生的，常见的错误类型和继承关系看这里：https://docs.python.org/2/library/exceptions.html#exception-hierarchy 使用 try...except 捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数 main() 调用 foo()，foo() 调用 bar()，结果 bar() 出错了，这时，只要 main() 捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

```

```
4  def bar(s):
5      return foo(s) * 2
6
7  def main():
8      try:
9          bar('0')
10     except StandardError, e:
11         print 'Error!'
12     finally:
13         print 'finally...'
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写 try...except...finally 的麻烦。调用堆栈如果错误没有被捕获，它就会一直往上抛，最后被 Python 解释器捕获，打印一个错误信息，然后程序退出。来看看 err.py：

```
1  # err.py:
2  def foo(s):
3      return 10 / int(s)
4
5  def bar(s):
6      return foo(s) * 2
7
8  def main():
9      bar('0')
10
11 main()
```

执行，结果如下：

```
1  $ python err.py
2  Traceback (most recent call last):
3    File "err.py", line 11, in <module>
4      main()
5    File "err.py", line 9, in main
6      bar('0')
7    File "err.py", line 6, in bar
8      return foo(s) * 2
9    File "err.py", line 3, in foo
10     return 10 / int(s)
11 ZeroDivisionError: integer division or modulo by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：错误信息第 1 行：Traceback (most recent call last):

告诉我们这是错误的跟踪信息。第 2 行：

```
1   File "err.py", line 11, in <module>
2     main()
```

调用 main() 出错了，在代码文件 err.py 的第 11 行代码，但原因是第 4 行：

```
File "err.py", line 9, in main
  bar('0')
```

调用 bar('0') 出错了，在代码文件 err.py 的第 9 行代码，但原因是第 6 行：

```
File "err.py", line 6, in bar
  return foo(s) * 2
```

原因是 return foo(s) * 2 这个语句出错了，但这还不是最终原因，继续往下看：

```
File "err.py", line 3, in foo
  return 10 / int(s)
```

原因是 return 10 / int(s) 这个语句出错了，这是错误产生的源头，因为下面打印了：ZeroDivisionError: integer division or modulo by zero

根据错误类型 ZeroDivisionError，我们判断，int(s) 本身并没有出错，但是 int(s) 返回 0，在计算 10 / 0 时出错，至此，找到错误源头。

记录错误

如果不捕获错误，自然可以让 Python 解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。Python 内置的 logging 模块可以非常容易地记录错误信息：

```
# err.py
import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        logging.exception(e)

main()
print 'END'
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python err.py
ERROR:root:integer division or modulo by zero
Traceback (most recent call last):
  File "err.py", line 12, in main
    bar('0')
```

```
  File "err.py", line 8, in bar
    return foo(s) * 2
  File "err.py", line 5, in foo
    return 10 / int(s)
ZeroDivisionError: integer division or modulo by zero
END
```

通过配置，logging 还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是 class，捕获一个错误就是捕获到该 class 的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python 的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。如果要抛出错误，首先根据需要，可以定义一个错误的 class，选择好继承关系，然后，用 raise 语句抛出一个错误的实例：

```
# err.py
class FooError(StandardError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python err.py
Traceback (most recent call last):
  ...
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择 Python 已有的内置的错误类型（比如 ValueError，TypeError），尽量使用 Python 内置的错误类型。最后，我们来看另一种错误处理的方式：

```
# err.py
def foo(s):
    n = int(s)
    return 10 / n

def bar(s):
    try:
        return foo(s) * 2
    except StandardError, e:
        print 'Error!'
        raise

def main():
```

```
    bar('0')

main()
```

在 bar() 函数中，我们明明已经捕获了错误，但是，打印一个 Error! 后，又把错误通过 raise 语句抛出去了，这不有病么？其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。raise 语句如果不带参数，就会把当前错误原样抛出。此外，在 except 中 raise 一个 Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input␣error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个 IOError 转换成毫不相干的 ValueError。小结 Python 内置的 try...except...finally 用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

调试

程序能一次写完并正常运行的概率很小，基本不超过 1%。总会有各种各样的 bug 需要修正。有的 bug 很简单，看看错误信息就知道，有的 bug 很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复 bug。第一种方法简单直接粗暴有效，就是用 print 把可能有问题的变量打印出来看看：

```
# err.py
def foo(s):
    n = int(s)
    print '>>>␣n␣=␣%d' % n
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

用 print 最大的坏处是将来还得删掉它，想想程序里到处都是 print，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。断言凡是用 print 来辅助查看的地方，都可以用断言（assert）来替代：

```
# err.py
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

assert 的意思是，表达式 n != 0 应该是 True，否则，后面的代码就会出错。如果断言失败，assert 语句本身就会抛出 AssertionError：

```
$ python err.py
Traceback (most recent call last):
  ...
AssertionError: n is zero!
```

程序中如果到处充斥着 assert，和 print 相比也好不到哪去。不过，启动 Python 解释器时可以用-O 参数来关闭 assert：

```
$ python -O err.py
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

关闭后，你可以把所有的 assert 语句当成 pass 来看。

logging 把 print 替换为 logging 是第 3 种方式，和 assert 比，logging 不会抛出错误，而且可以输出到文件：

```
# err.py
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print 10 / n
```

logging.info() 就可以输出一段文本。运行，发现除了 ZeroDivisionError，没有任何信息。怎么回事？别急，在 import logging 之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这就是 logging 的好处，它允许你指定记录信息的级别，有 debug，info，warning，error 等几个级别，当我们指定 level=INFO 时，logging.debug 就不起作用了。同理，指定 level=WARNING 后，debug 和 info 就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。logging 的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如 console 和文件。pdb 第 4 种方式是启动 Python 的调试器 pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print 10 / n
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/sicp/err.py(2)<module>()
-> s = '0'
```

以参数-m pdb 启动后，pdb 定位到下一步要执行的代码-> s = ’0’。输入命令 l 来查看代码：

```
(Pdb) l
  1 # err.py
  2 -> s = '0'
  3 n = int(s)
  4 print 10 / n
[EOF]
```

输入命令 n 可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/sicp/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
```

任何时候都可以输入命令 p 变量名来查看变量：

```
(Pdb) p s
```

```
'0'
(Pdb) p n
0
```

输入命令 q 结束调试，退出程序：

```
(Pdb) n
ZeroDivisionError: 'integer division or modulo by zero'
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
(Pdb) q
```

这种通过 pdb 在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第 999 行得敲多少命令啊。还好，我们还有另一种调试方法。pdb.set_trace() 这个方法也是用 pdb，但是不需要单步执行，我们只需要 import pdb，然后，在可能出错的地方放一个 pdb.set_trace()，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print 10 / n
```

运行代码，程序会自动在 pdb.set_trace() 暂停并进入 pdb 调试环境，可以用命令 p 查看变量，或者用命令 c 继续运行：

```
$ python err.py
> /Users/michael/Github/sicp/err.py(7)<module>()
-> print 10 / n
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这个方式比直接启动 pdb 单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的 IDE。目前比较好的 Python IDE 有 PyCharm：http://www.jetbrains.com/pycharm/ 另外，Eclipse 加上 pydev 插件也可以调试 Python 程序。小结写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。虽然用 IDE 调试起来比较方便，但是最后你会发现，logging 才是终极武器。

3.2.25 函数与自变量 (参数)

在 python 中过程就是函数，在 fortran 中虽然有 function 和 subroutine 的区别，但本质没有太多的差别，只是 function 通常有一个该函数类型的返回值。

python 中的函数返回三种类型的对象:none，obj，tuple

函数的参数有:

positional_args，keyword_args，*tuple_grp_nonkw_args，**dict_grp_kw_args

调用的时候，是两种输入参数方式，一种是直接给值，一种是 key= 值的方式，参数非常自由，有 key 先取 key，然后再去其他。

函数可以利用 ___doc___ 设置 doc 内容

函数可以利用 tester 设置测试代码。

函数名也可以作为参数，传递给其它函数，便于在其内进行调用。

函数式编程主要包括 lambda 和内建函数两类，包括 apply 已经被变参数元组和字典取代。filter(func,seq) 可以用列表解析取代。map(func,seq1,seq2)=[func(seq1[0],seq2[0]),...]。reduce(func,seq)=func(func(func(seq[0],seq[1]),seq[2]),seq[3])。

比如 reduce((lambda x,y:x+y),range(5))=10。

注意 python3 中 reduce 已经放到 functools 模块中了，而且 functools 模块中还有不少函数式编程的工具，比如 partial 等。

partial 可以做函数调用模板，可以省略一些参数。

如果函数包含了对其自身的调用，那么该函数就是递归的。

函数的参数

定义模块级的参数用 global。Global 声明把赋值的名字映射到一个包含它的模块的作用域中。

参数的传递: 对于不可变参数，类似于 c 中的传值对于可变参数，类似于 c 中的传指针

函数设计注意点: 1. 输入用参数，输出用 return 2. 只在绝对必要时采用全局变量 3. 不要改变参数除非调用者期望这样做

```
1  print #空输出空一行
2  print '----------------------------------'
3  print '函数和函数的参数:'
4  print '----------------------------------'
5  a=abs #函数可以赋值给变量
6  print a(-1)
7
8  def nop():
9      pass #pass表示什么都不做
10 print nop()
11
12 def my_abs(x):
```

```
13      if not isinstance(x, (int, float)): #用内置函数isinstance实现可以类型检查
14          raise TypeError('输入参数数据类型错误!')
15      if x >= 0:
16          return x #返回参数用return,当没有return时自动返回None
17      else:
18          return -x
19  print my_abs(-1.695)
20  print my_abs(+6.695)
21  #print my_abs('a')
22
23  import math #使用数学函数,用math
24  def move(x, y, step, angle=0):
25      nx = x + step * math.cos(angle)
26      ny = y + step * math.sin(angle)
27      return nx, ny
28  x,y=move(0,0,1,math.pi/6) #pi 用math.pi表示
29  a=move(0,0,1,math.pi/6) #返回多个值本质上是返回一个tuple
30  print 'x=',x
31  print 'y=',y
32  print 'a=',a
33
34  def power(x, n=2): #当使用有默认值的参数时,默认参数可以不给出,当然默认参数必须放在参数列表的最后。
35      s = 1
36      while n > 0:
37          n = n - 1
38          s = s * x
39      return s
40  print 'power(5)=',power(5)
41  print 'power(5,2)=',power(5,2)
42  print 'power(5,1)=',power(5,1)
43  print 'power(5,4)=',power(5,4)
44
45
46  x,y=0,0 #当使用有多个默认值的参数时,可以按顺序的给出参数,如果不按顺序,应给出参数的名
47  r=20
48  # draw_circle(x, y, r)
49  # draw_circle(0, 0, 20, linecolor=0xff0000)
50  # draw_circle(0, 0, 20, linecolor=0xff0000, penwidth=5)
51  # draw_circle(0, 0, 20, linecolor=0xff0000, fillcolor=0xffff00, penwidth=5)
52
53  #定义默认参数要牢记一点：默认参数必须指向不变对象!
54  #空对象可以用None这个不变对象来实现：
55  def add_end(L=None):
56      if L is None:
57          L = []
58      L.append('END')
59      return L
60  print 'L=',add_end([1, 2, 3])
61  print 'L=',add_end(['x', 'y', 'z'])
62  print 'L=',add_end()
```

```
63  print 'L=',add_end()
64
65  def write_list(L=None):
66      if L is None:
67          print L
68      else:
69          print 'len(L)=',len(L)
70          for i in range(len(L)):
71              print i,L[i]
72  write_list()
73  write_list([1, 2, 3])
74  write_list(['x', 'y', 'z'])
75  print 'range(3):',range(3) #需要注意range给出范围是到终点值−1的值
76  print 'range(1,4,1):',range(1,4,1)
77
78  #还可以定义可变参数，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。
79
80  #输入参数可以是list或tuple,这样参数是一个,但实际信息可以是多个
81  #其本质其实是固定参数,但这个参数因为可以是list或者tuple所以实际信息可以多个
82  def calc(numbers):
83      sum = 0
84      for n in numbers:
85          sum = sum + n * n
86      return sum
87  #print calc() #numbers是一个参数,不给出是错误的
88  #print calc(None) #用None当然也是错误的
89  print calc([]) #但用空list或空tuple是可以的
90  print calc(())
91  a=[1,2,3]
92  print calc(a)
93  b=(1,2,3,4,5)
94  print calc(b)
95
96  #真正的可变参数用*var来表示:
97  #可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple
98  def calca(*numbers):
99      sum = 0
100     for n in numbers:
101         sum = sum + n * n
102     return sum
103 #print calca([]) #输入不能再使用不可变的形式
104 #print calca(())
105 #print calca(a)
106 #print calca(b)
107 #输入应使用可变的形式
108 print calca() #numbers是可变参数
109 #print calca(None) #用None仍然不行
110 print calca(1,2,3) #输入,直接给出可变个数的参数
111 print calca(1,2,3,4,5)
112 #可以先组装出一个list或者tuple，然后，把它用带*号的方法转换为可变参数后输入
```

```
113 print calca(*a) #利用*list,*tuple也是可以的
114 print calca(*b)
115
116 #关键子参数用**var表示
117 #关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict
118
119 #关键字参数有什么用？它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，
120 #但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是
121 #必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。
122 def person(name, age, **kw):
123     print 'name:', name, 'age:', age, 'other:', kw
124 person('Michael', 30)
125 person('Bob', 35, city='Beijing') #注意关键字参数的输入方式
126 person('Adam', 45, gender='M', job='Engineer')
127 #可以先组装出一个dict，然后，把该dict用带**号的方法转换为关键字参数后输入
128 kw = {'city': 'Beijing', 'job': 'Engineer'}
129 person('Jack', 24, **kw)
130
131 #定义函数可以用不同参数组合，参数列表可以包括:必选参数、默认参数、可变参数和关键字参数，
132 #这4种参数都可以一起使用，或者只用其中某些，但是请注意，
133 #参数定义的顺序必须是：必选参数、默认参数、可变参数和关键字参数。
134 def func(a, b, c=0,d=0, *args, **kw):
135     print 'a␣=', a, 'b␣=', b, 'c␣=', c,'d␣=', d, 'args␣=', args, 'kw␣=', kw
136
137 func(1, 2)
138 func(1, 2,3)
139 func(1, 2,3)
140 func(1, 2,3,'a','b')
141 func(1, 2,3,'a','b',x=99,y=33)
142 #func(1, 2,'a','b',x=99,c=3) #注意组合使用时,各类参数的顺序不能调换
143 func(1, 2,c=3)
144 func(1, 2,d=2,c=3)
145 #func(1, 2,d=2,c=3,'a') #一旦使用了关键字参数后,不能再后面使用位置参数了
146 #func(1, 2,3,d=3,'a','b') #若要使用默认参数的关键字输入,其后不能再使用可变参数和关键字参数。
147 #func(1, 2,3,d=3 ,'a','b', x=99) #
148 args=('a','b')
149 kw = {'x': 99}
150 #func(1, 2,d=2,c=3,*args) #组装好的可变参数不行
151 func(1, 2,d=2,c=3,x=66) #再使用关键参数也是可以的。
152 func(1, 2,d=2,c=3,**kw) #使用组装好的关键字参数是可以的
```

递归函数

```
1 print #空输出空一行
2 print '---------------------------------'
3 print '递归函数:'
4 print '---------------------------------'
5
6 def fact(n):
```

```
7       if n==1:
8           return 1
9       else:
10          return n*fact(n-1)
11
12  print fact(1)
13  print fact(5)
14  print fact(20)
15  #print fact(1000) #递归层太多,栈不足
16
17  #单纯递归是有栈大小限制的,所以可以利用尾递归的方式
18  #这种方式类似于fortran中的过程递归
19  #return也具有类似fortran中的call过程的功能
20  #注意尾递归的本质是循环,但有的时候处理一些特殊问题可能好理解一些
21  def factb(n):
22      return factw(1,n)
23
24  def factw(p,n):
25      p=n*p
26      m=n-1
27      if m<=1:
28          return p
29      else:
30          return factw(p,m)
31
32  print factw(1,20)
33  print factw(1,1000)
34  #print factb(1000) #python语言并没有对尾递归做优化,如上的factb函数仍然会栈溢出
```

3.2.26 变量的作用域 (范围)

全局作用域-> 最高级别的变量-> 除非被删除，否则存货到脚本运行结束

名字在过程之内声明-> 局部变量-> 一旦函数完成，框架被释放，变量将会离开作用域。

明确一个已命名的变量为全局变量，必须使用 global 语句

除了全局变量和局部变量以外的变量称为自由变量

在一个内部函数里面，定义一个内部函数，对其外部作用域 (但不是全局作用域) 的变量进行引用，那么该内部函数认为是闭包。

静态嵌套域

3.2.27 名称空间

搜索路径在 sys.path 中

首先加载内建名称空间，有 ___builtins___(不同于 ___builtin___) 模块中的名字构成

随后加载执行模块的全局名称空间，它会在模块开始执行后变为活动名称空间

如果在执行时，调用一个函数，那么将创建出第三个名称空间，即局部名称空间
可以通过 globals()，local() 函数判断某一个名字属于哪个名称空间
globals() 和 local() 返回返回调用者全局和局部名称空间字典
在任何需要放置数据的地方都可以获得一个名称空间，比如在任何时候给函数添加属性。
一个实例就是一个名称空间。

3.2.28 函数式编程

函数式编程-高阶函数, 排序 Map，reduce，filter 函数提供了比较方便的功能，但这不是必须的，用函数也是很容易实现的。

比如: filter(function, iterable) is equivalent to the generator expression (item for item in iterable if function(item)) if function is not None and (item for item in iterable if item) if function is None.

```
1
2  print #空输出空一行
3  print '----------------------------------'
4  print '函数式编程-高阶函数,排序,'
5  print '----------------------------------'
6
7  #map()函数这种能够接收函数作为参数的函数，称之为高阶函数（Higher−order function）。
8  def sq(x):
9      return x*x
10 print map(sq, [1, 2, 3, 4, 5, 6, 7, 8, 9])
11
12 #reduce把一个函数作用在一个序列[x1, x2, x3...]上，这个函数必须接收两个参数，
13 #reduce把结果继续和序列的下一个元素做累积计算
14 def add(x, y):
15     return 10*x+y
16 print reduce(add, [1, 2, 3, 4, 5, 6, 7, 8, 9])
17
18 def str2int(s):
19     return reduce(lambda x,y: x*10+y, map(int, s))
20 print str2int('96538')
21
22
23 #通常规定，对于两个元素x和y，如果认为x < y，则返回−1，如果认为x == y，则返回0，
24 #如果认为x > y，则返回1，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序
25 print sorted([36, 5, 12, 9, 21])
26 def reversed_cmp(x, y):
27     if x > y:
28         return −1
29     if x < y:
30         return 1
31     return 0
32 print sorted([36, 5, 12, 9, 21], reversed_cmp)
33 def cmp_ignore_case(s1, s2):
```

```
34      u1 = s1.upper()
35      u2 = s2.upper()
36      if u1 < u2:
37          return -1
38      if u1 > u2:
39          return 1
40      return 0
41  print sorted(['about', 'bob', 'Zoo', 'Credit'])
42  print sorted(['about', 'bob', 'Zoo', 'Credit'], cmp_ignore_case)
43
44  #高阶函数还可以返回函数
45  def lazy_sum(*args):
46      def sum():
47          ax = 0
48          for n in args:
49              ax = ax + n
50          return ax
51      return sum
52  f=lazy_sum(1, 3, 5, 7, 9)
53  print f #这是定义
54  print f() #这是函数调用
55
56  f1 = lazy_sum(1, 3, 5, 7, 9)
57  f2 = lazy_sum(1, 3, 5, 7, 9)
58  print f1==f2 #每次生成的函数都是不同的
59  print f1()==f2()
60
61  def mymap(myfun, myvar): #自定义的map函数
62      resulta=[]
63      for var in myvar:
64          resulta.append(myfun(var))
65      return resulta
66
67  print mymap(sq, [1, 2, 3, 4, 5, 6, 7, 8, 9])
68  print map(sq, [1, 2, 3, 4, 5, 6, 7, 8, 9])
69
70  def fstr(x):
71      return x
72  print mymap(fstr, "abcd")
73  print map(fstr, "abcd")
74
75  #在传入函数时，有时不需要显式地定义函数，直接传入匿名函数更方便
76  #lambda x: x*x #等价于
77  #def f(x):
78  # return x * x
79
80  print map(lambda x: x*x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
81
82  #匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。
83  #用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。
```

```
#匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数
fa = lambda x: x * x
print fa
print fa(5)

#也可以把匿名函数作为返回值返回，比如：
def build(x, y):
    return lambda: x * x + y * y

print build(5, 6)
fb=build(5, 6)
print fb()
```

3.2.29 纯函数和修饰器

函数会建立一个新的 Time 对象，初始化它的属性，并传回这个新对象的参照。这称为一个纯函数，因为它不会变更任何传给它作为自变量的对象，而且没有副作用，例如显示一个值或是取得使用者输入。

有时候，一个函数可以变更一个或多个它接收作为自变量的对象是很有用的。通常，呼叫者会保留它所传出对象的参照，所以呼叫者可以看到任何函数所进行的变更。这种函数就称为修饰子。

任何可以用修饰子完成的事都可以用纯函数来完成。事实上，一些程序语言只允许纯函数。证据显示使用纯函数的程序比使用修饰子的程序开发速度较快，错误也较少。不过，修饰子有时是很方便的，在某些情况下，函数程序则较没有效率。总体来说，我们建议你在合理时撰写纯函数，只在有利时才采用修饰子。这种方法可以称为函数式程序风格（functional programming style）。

装饰器有点像 reduce 的效果，比如:

```
@deco2
@deco1
def func:pass

等价于:
def func:pass
func=deco2(deco1(func))
```

装饰器如果带参数，比如:

```
@deco1(deco_args)
def func():pass

等价于:
def func():pass
func=deco1(deco_args)(func)#deco1先返回一个函数然后修饰foo
```

```
print #空输出空一行
print '----------------------------------'
print '函数装饰器'
print '----------------------------------'

import functools #用于@functools.wraps(func)函数

def loga(func): #定义一个装饰器,loga其对func函数进行装饰
    @functools.wraps(func) #把wrapper函数的名字属性变成与func函数一致
    def wrapper(*args, **kw): #把原函数包含起来,用wrapper函数传递变量
        print 'call %s():' % func.__name__ #函数的名字属性
        return func(*args, **kw)
    return wrapper

def logb(text): #定义一个装饰器,logb可以输入参数text
    def decorator(func): #定义一个装饰器,decorator用于输入原函数名
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator

@loga
def todaya():
    print '2016-03-09'
print todaya.__name__
print todaya()

@logb('excute:')
def todayb():
    print '2016-03-09'
print todayb.__name__
print todayb()

def logc(func): #定义一个装饰器,loga其对func函数进行装饰
    @functools.wraps(func) #把wrapper函数的名字属性变成与func函数一致
    def wrapper(*args, **kw): #把原函数包含起来,用wrapper函数传递变量
        print 'begin call function:'
        print '%s():' % func.__name__ #函数的名字属性
        print 'end call function. result:'
        return func(*args, **kw)
    return wrapper

@logc
def todayc():
    print '2016-03-09'
print todayc.__name__
print todayc()
```

```
50
51 #定义@log,既支持@log,又支持@log('execute'),貌似有问题,以后更熟悉了再来考虑
52 def logd(*text): #定义一个装饰器,logb可以输入参数text
53     def decorator(func):
54         @functools.wraps(func)
55         def wrapper(*args, **kw):
56             print '%s %s():' %(text, func.__name__)
57             return func(*args, **kw)
58         return wrapper
59     return decorator
60 @logd()
61 def todayd():
62     print '2016-03-09'
63 print todayd.__name__
64 print todayd()
65 @logd('call ')
66 def todaye():
67     print '2016-03-09'
68 print todaye.__name__
69 print todaye()
```

3.2.30 生成器

挂起返回中间值并多次继续的协调程序称为生成器。

句法上讲，生成器是一个带 yield 的语句函数。

一个函数或子程序仅返回一次，但一个生成器能暂停执行并返回一个中间结果，这是 yield 语句的功能，返回一个值给调用者并暂停执行。当生成器的 next 方法被调用时，它会准确的从离开的地方继续。

生成器有 next，send，close 方法。

迭代器需要利用 ___iter___，next() 构造。

生成器和迭代器的区别？

迭代器是让程序员可以遍历一个容器 (特别是列表) 的对象。一个可迭代对象时 python 中的任意对象，只要它定义了可以返回一个迭代器的 ___iter___ 方法，或者定义了可以支持下标索引的 ___getitem___ 方法。一个迭代器是任意一个对象，只要它定义了一个 ___next___ 方法。当使用一个循环来遍历一个东西时，这个过程称为迭代。生成器是一种迭代器，但只能对其迭代一次，因为他们没有把所有的值都存在内存中，而是在运行时生成值。使用它们，要么通过使用一个 for 循环，要么将其传递给任意可以进行迭代的函数或结构。大多数时候哦，迭代器是用函数实现的。

str 对象时一个可迭代对象，但不是一个迭代器，如果需要对其进行迭代，那么需要用 iter 函数，它将根据一个可迭代对象返回一个迭代器对象。

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

```
#_*_coding: utf-8 _*_

"""
测试-生成器
"""

import os
import sys
import subprocess
import re


def gentest():
    #eg from intermediate python

    #
    str1="yasuoob"
    for char in str1:
        print(char)
    #print(next(str1))#error

    #iterator
    print("\n")
    myiter=iter(str1)
    for i in range(len(str1)):
        print(next(myiter))#right

def main():
    print("测试程序!")
    print(os.getcwd())#路径问题
    gentest()
    pass

if ___name___ == "__main__":
    main()
```

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

"""
测试迭代器和生成器
"""

import os
import codecs

def maxlinelensa():
    f=codecs.open("./baa.dat","r",encoding="utf-8")
    alllinelens=[len(x.strip()) for x in f.readlines()] #列表解析
    f.close()
```

```
15      return max(alllinelens)
16
17  def maxlinelensb():
18      f=codecs.open("./baa.dat","r",encoding="utf-8")
19      alllinelens=[len(x.strip()) for x in f] #列表解析，文件就是迭代器
20      f.close()
21      return max(alllinelens)
22
23  def maxlinelensc():
24      f=codecs.open("./baa.dat","r",encoding="utf-8")
25      longest=max(len(x.strip()) for x in f) #生成器
26      f.close()
27      return longest
28
29  def maxlinelensd():
30      return max(len(x.strip()) for x in codecs.open("./baa.dat","r",encoding="utf-8")) #生成器
31
32
33  def main():
34      print("a=",maxlinelensa())
35      print("a=",maxlinelensb())
36      print("a=",maxlinelensc())
37      print("a=",maxlinelensd())
38      print("测试程序!")
39      pass
40
41  if __name__ == "__main__":
42      main()
```

3.2.31 自身

自省是指代码可以查看内存中以对象形式存在的其它模块和函数，获取它们的信息，并对它们进行操作。用这种方法，你可以定义没有名称的函数，不按函数声明的参数顺序调用函数，甚至引用事先并不知道名称的函数。

type 、str、dir，callable ，getattr 函数 type 函数返回任意对象的数据类型。在 types 模块中列出了可能的数据类型。这对于处理多种数据类型的帮助者函数非常有用。str 将数据强制转换为字符串。每种数据类型都可以强制转换为字符串。dir 函数返回任意对象的属性和方法列表，包括模块对象、函数对象、字符串对象、列表对象、字典对象 callable(object) Return True if the object argument appears callable, False if not. If this returns true, it is still possible that a call fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a __call__() method. getattr 获取对象引用，Python 函数是对象。使用 getattr 函数，可以得到一个直到运行时才知道名称的函数的引用。getattr 不仅仅适用于内置数据类型，也可作用于模块。getattr 常见的使用模式是作为一个分发者。举个例子，如果你有一个程序可

以以不同的格式输出数据，你可以为每种输出格式定义各自的格式输出函数，然后使用唯一的分发函数调用所需的格式输出函数。例如，让我们假设有一个以 HTML、XML 和普通文本格式打印站点统计的程序。输出格式在命令行中指定，或者保存在配置文件中。statsout 模块定义了三个函数：output_html、output_xml 和 output_text。然后主程序定义了唯一的输出函数。getattr 能够使用可选的第三个参数，一个缺省返回值。比如:

```
import statsout
def output(data, format="text"):
output_function = getattr(statsout, "output_%s" % format, statsout.output_text)
return output_function(data) (1)
```

这个函数调用一定可以工作，因为你在调用 getattr 时添加了第三个参数。第三个参数是一个缺省返回值，如果第二个参数指定的属性或者方法没能找到，则将返回这个缺省返回值。getattr 是相当强大的。它是自省的核心。

3.2.32 lambda 函数

快速定义单行的最小函数。这些叫做 lambda 的函数，是从 Lisp 借用来的，可以用在任何需要函数的地方。lambda 函数可以接收任意多个参数 (包括可选参数) 并且返回单个表达式的值。lambda 函数不能包含命令，包含的表达式不能超过一个。

注意: lambda 是一个表达式，而不是一个语句 Lambda 的结构体是一个单一结构体，不是一个语句块

```
#!/usr/bin/env python3
#_*_coding: utf−8 _*_


def func(x,y,z):return x+y+z
print(func(1,2,3))

f=lambda x,y,z:x+y+z
print(f(1,2,3))
```

Python3 中 apply 内置函数已经取消了。

3.2.33 执行环境

input 把输入作为 python 表达式求值

eval 接受引号内的字符串并把它作为 python 表达式求值。计算存储在字符串中有效 python 表达式。

repr 用了取得对象的规范字符串表示,基本上用来获取对象的可打印表示形式 eval(repr(obj))==obj

exec 可以读取脚本内容并执行。执行存储在字符串或文件中的 python 语句。

execfile 直接调用 python 脚本文件执行，python3 中已经不用该命令。

os 有执行外部程序的接口，subprocess 模块代替 os，主要是两个函数，call，popen

退出执行用 sys.exit()

利用 cpickle 模块可以把 python 对象保存到文件以便后用。

3.2.34 类和对象

旧式类-python 诞生时创造的类，没有指定父类

新式类-types 和 classes 统一为类，必须继承一个父类，默认是 object 类

从类创建一个实例的过程称为实例化。

最简单的情况，类仅作为名称空间。属性可以是类的，也可以使实例的，实例的属性实质上是动态的，无需预先声明或赋值。方法定义在类中，只能被实例代用。self 参数代表实例本身，实例方法需要实例 self，而静态方法或类方法不需要，类方法需要类而不是实例。当一个类被实例化，可以利用 ___init___ 定义额外的行为。

创建一个之类，如果需要，每个子类最好定义它自己的构造器。不然，基类的构造器会被调用。如果子类重写了构造器，那么基类的构造器必须要显示的写出才会被执行。

抽象是对现实世界问题和实体的本质表现，行为和特征建模。建立一个相关关系，用于描绘程序结构，从而实现这种模型，抽象包括模型的数据属性，还定义了数据的接口，抽象的实现是对数据及其接口的现实化。

封装描述了对数据和信息进行隐藏的观念，它对数据属性提供接口和访问函数。对数据提供接口，以便进行不规范的数据访问。接口简单说是提供访问数据属性的函数。

合成 (组合)，多个不同的类合成一个大的类。

比如:

```
1  class A:
2      pass
3
4  class B:
5      pass
6
7  class C:
8      def ___init___(self,a,b):
9          self.va=A(a)
10         self.vb=B(b)
```

派生描述子类的创建。继承描述子类属性从祖先类继承这种昂视。继承结构表示多“代”派生。泛化表示所有子类和父类及祖先类一样的特征。特化表示所有子类的定义，即与祖先类不同的属性。

比如:

```
1  class A:
2      pass
3
```

```
4 class B:
5     pass
6
7 class C(A,B):
8     pass
```

多态表明动态 (运行时) 绑定的存在，运行重载及运行时类型确定和验证。对象通过共同的属性和动作来按字数行为进行操作和访问。

自省表示赋予程序员进行“手工类型检查”的工作，也称反射。展示某对象如何在运行期间取得自身的信息。

类是一种数据结构，用它来定义对象。对象把数据属性和行为特征融合在一起。类是现实世界的抽象的实体以编程像是出现。实例是对象的具体化。类是蓝图或模型，用来产生真实的物体 (实例)。创建一个类，实际上是创建一个数据类型。

类属性与类绑定，与任何实例无关，对于所有实例都是固定的，就是静态数据，类似与 c++ 中变量声明前加上 static 关键字。常用来跟踪与类相关的值。

多数情况下，会使用实例属性，而不是类属性。类属性用类名调用。类可以用 dir，类的 ___dict___ 属性，或 vars(类对象) 查。

特殊类属性包括:___name___,___doc___,___bases___,___dict___,___module___,___class___。

特殊方法包括:init，new，del 等，定制类的一些特殊方法，可以模拟标准类型，也可以重载操作符。

类是一种数据结构定义类型，那么实例则声明了一个这种类型的变量，调用类可以创建实例。

实例属性，也可以用 dir(实例对象)，实例的 ___dict___ 属性，或 vars(实例对象) 查。

类和实例都是名字空间，类是类属性的名字空间，实例是实例属性的名字空间。可以用类来访问类属性，如果实例没有同名属性，也可以用实例访问。

绑定，当存在一个实例时，方法才被认为绑定到实例上。没有实例的方法就是未绑定的。任何方法的第一个参数都是 self，表示调用此方法的实例对象。当需要调用非绑定方法的时候，必须传递 self 参数。

静态方法和类方法，静态方法是类中的函数，而类方法需要类而不是实例作为第一个参数。需要用 staticmethod，或 classmethod 函数声明，或者利用函数修饰器 staticmethod，或 classmethod 进行修饰，调用时可以通过类或实例调用。

super 函数用于调用基类的函数。

多重继承源自一个基类。

设置 ___slot___ 属性后，该属性中的实例属性访问才不会出现异常

一些相关函数包括: issubclass，isinstance，hasattr，setattr，getattr，delattr，dir，vars，super

pass

实例属性可以自由添加

对象可以作为函数参数

```
class Point:
    pass

class Rectangle:
    pass

box = Rectangle()
box.width = 100.0
box.height = 200.0
print(box)
print(box.width )
print(box.height)

box.corner=Point()
box.corner.x=100
box.corner.y=200
print(box.corner)
print(box.corner.x )
print(box.corner.y )
```

类的方法的参数可以是用 self，也可以不用。要更具有面向对象的风格，那么应该使用 self，也需要使用初始化函数，并进行参数初始化

运算符可以重载

```
def __add__(self, other):
return Point(self.x + other.x, self.y + other.y)
```

类的对象要用 self 做前缀引用，而一个类的公用变量用类名做前缀引用。类的公用变量如同 c++/java 的 static 变量一样，被类的所有实例共享。构造函数，init 在类实例化时自动调用。而实例删除时，自动调用析构函数 del。

举个例子:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

class base:
    def __init__(self):
        self.data=[]

    def add(self,x):
        self.data.append(x)

class child(base):
    def plus(self,a,b):
        return a+b;

```

```
child1=child()
child1.add("str1")
print (child1.data)
print (child1.plus(2,3))
```

再举个例子:

```
class member:
    count=0
    def ___init___(self,name,age):
        self.name=name
        self.age=age
        member.count+=1
        print("(initialized␣menber:␣%s)" % self.name)

    def tell(self):
        print("name:%s␣age:%s" %(self.name,self.age),end='')

class teacher(member):
    def ___init___(self,name,age,salary):
        member.___init___(self,name,age)
        self.salary=salary
        print("(initialized␣teacher:␣%s)" % self.name)

    def tell(self):
        member.tell(self)
        print("␣salary:%d" %(self.salary))

class student(member):
    def ___init___(self,name,age,marks):
        member.___init___(self,name,age)
        self.marks=[]
        if(isinstance(marks,list)):
            self.marks.extend(marks)
        else:
            self.marks.append(marks)

        # self.marks=marks
        print("(initialized␣student:␣%s)" % self.name)

    def tell(self):
        member.tell(self)
        print("␣marks:","␣".join(["%s" % elem for elem in self.marks]))
        #print(" marks:",self.marks)


t1=teacher("zs",40,3000)
s1=student('ls',23,[77,55])
s2=student('ww',23,[90,75,86])
s3=student('qf',23,81)

```

```
45 print()
46 print(member.count)
47 members=[t1,s1,s2,s3]
48 for men in members:
49     men.tell()
```

特殊方法

特殊方法可以模仿某个行为，比如: init，del，str，lt，getitem，len 等。

3.2.35 对象的复制

copy 模块可以进行对象的浅层和深层复制: copy.copy(x) Return a shallow copy of x. copy.deepcopy(x) Return a deep copy of x.

3.2.36 docstring 和单元测试

3.2.37 pdb 调试

3.2.38 模块和包

python 的模块级 py 文件可以定义在包里面。包是一个文件夹，文件夹下有一个文件 ___init___.py。如果包下面还有一些子文件夹，同样的，子文件夹下面放文件 ___init___.py 以及模块文件。

模块导入:

from module import * 可以把模块内的所有函数引入，使用时不用加模块前缀

import mod1,mod2... 导入模块

from mod import name1,name2,... 把指定名称道路当前作用域

import mod as md 导入同时指定局部绑定名称

包导入:

import phone(包).mobile(子包).anlog(模块)

from phone import mobile

form phone.mobile import anlog

from phone.mobile.anlog import dial(函数名)

sys.modules 包含当前载入摸块构成的字典。

_ 开头的私有属性，除非显式导入，否则不会被导入。

模块引入有两种方式: import UserDict from UserDict import UserDict 第二种直接引入命名空间，因此不需要要加模块 (对象，python 中任何事物都是对象，模块也是对象) 限定，即不需要加点。Python 中的 from module import * 像 Perl 中的 use module ； Python 中的

import module 像 Perl 中的 require module 。使用 from module import？• 如果你要经常访问模块的属性和方法，且不想一遍又一遍地敲入模块名，使用 from module import。• 如果你想要有选择地导入某些属性和方法，而不想要其它的，使用 from module import。• 如果模块包含的属性和方法与你的某个模块同名，你必须使用 import module 来避免名字冲突。尽量少用 from module import * ，因为判定一个特殊的函数或属性是从哪来的有些困难，并且会造成调试和重构都更困难。

与其它任何 Python 的东西一样，模块也是对象。只要导入了，总可以用全局 dictionary sys.modules 来得到一个模块的引用。sys 模块包含了系统级的信息，像正在运行的 Python 的版本 (sys.version 或 sys.version_info)，和系统级选项，像最大允许递归的深度 (sys.getrecursionlimit() 和 sys.setrecursionlimit())。sys.modules 是一个字典，它包含了从 Python 开始运行起，被导入的所有模块。键字就是模块名，键值就是模块对象。请注意除了你的程序导入的模块外还有其它模块。Python 在启动时预先装入了一些模块，如果你在一个 Python IDE 环境下，sys.modules 包含了你在 IDE 中运行的所有程序所导入的所有模块。

每个 Python 类都拥有一个内置的类属性 ___module___，它定义了这个类的模块的名字。将它与 sys.modules 字典复合使用，你可以得到定义了某个类的模块的引用。

3.2.39 开发方法

渐进式开发（incremental development）的技术，渐进式开发的目标是藉由一次只加入和测试少量程序代码，避免冗长的除虫时间。

原型开发（prototype development）的的程序开发方法。针对每一个情况，我们撰写了一个粗略版本（或称原型），让它可以执行基本运算，然后我们在一些情况下测试，并在找到暇疵时更正它。虽然这种方法可以很有效率，不过却可能导致程序代码不必要的复杂因为它处理许多特别的情况而且也不太可靠因为你很难知道你是否已经找到所有的错误。

另一种方法则是计划式开发（planned development），在这种开发方式中，对问题的高度理解可以使开发更为容易。

3.2.40 网页应用

数据库

服务器

网页表单熟悉保单的最好方法是写一个可以接受表单数据的程序。web 接受数据可以有两种方式 1 是从浏览器的地址栏输入 2 是从网页创建的表单输入

网页框架

3.2.41 接球游戏

考虑球的运动，运动范围，从边界返回，响应键盘/鼠标，移动手套，显示得分。

3.3 面向对象编程

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成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个 dict 表示：

```
std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

```
def print_score(std):
    print '%s:␣%s' % (std['name'], std['score'])
```

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 和 score 这两个属性（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 print_score 消息，让对象自己把自己的数据打印出来。

```
class Student(object):

    def ___init___(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print '%s:␣%s' % (self.name, self.score)
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart␣Simpson', 98)
lisa = Student('Lisa␣Simpson', 77)
bart.print_score()
```

```
4 lisa.print_score()
```

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。小结数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

3.3.1 类和实例

Python 中的 pass 语句就像 Java 或 C 中的大括号空集 ()。Python 的类没有显示的构造函数和析构函数。Python 类的确存在与构造函数相似的东西：___init___ 方法。每个类方法的第一个参数，包括 ___init___，都是指向类的当前实例的引用。按照习惯这个参数总是被称为 self。在 ___init___ 方法中，self 指向新创建的对象；在其它的类方法中，它指向方法被调用的类实例。尽管当定义方法时你需要明确指定 self，但在调用方法时，你不用指定它，Python 会替你自动加上的。___init___ 方法可以接受任意数目的参数，就像函数一样，参数可以用缺省值定义，即可以设置成对于调用者可选。在本例中，filename 有一个缺省值 None，即 Python 的空值。任何 Python 类方法的第一个参数 (对当前实例的引用) 都叫做 self。这个参数扮演着 C++ 或 Java 中的保留字 this 的角色，但 self 在 Python 中并不是一个保留字，它只是一个命名习惯。虽然如此，也请除了 self 之外不要使用其它的名字，这是一个非常坚固的习惯。___init___ 方法是可选的，但是一旦你定义了，就必须记得显示调用父类的 ___init___ 方法 (如果它定义了的话)。这样更是正确的：无论何时子类想扩展父类的行为，后代方法必须在适当的时机，使用适当的参数，显式调用父类方法。

在 Python 中，创建类的实例只要调用一个类，仿佛它是一个函数就行了。不像 C++ 或 Java 有一个明确的 new 操作符。如果说创建一个新的实例是容易的，那么销毁它们甚至更容易。通常，不需要明确地释放实例，因为当指派给它们的变量超出作用域时，它们会被自动地释放。内存泄漏在 Python 中很少见。

Python 两种都不支持，总之是没有任何形式的函数重载。一个 ___init___ 方法就是一个 ___init___ 方法，不管它有什么样的参数。每个类只能有一个 ___init___ 方法，并且如果一个子类拥有一个 ___init___ 方法，它总是覆盖父类的 ___init___ 方法，甚至子类可以用不同的参数列表来定义它。

应该总是在 ___init___ 方法中给一个实例的所有数据属性赋予一个初始值。这样做将会节省你在后面调试的时间，不必为捕捉因使用未初始化 (也就是不存在) 的属性而导致的 AttributeError 异常费时费力。

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，比如 Student 类，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。仍以 Student 类为例，在 Python 中，定义类是通过 class 关键字：

```
class Student(object):
    pass
```

class 后面紧接着是类名，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的，继承的概念我们后面再讲，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。

定义好了 Student 类，就可以根据 Student 类创建出 Student 的实例，创建实例是通过类名 +() 实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量 bart 指向的就是一个 Student 的 object，后面的 0x10a67a590 是内存地址，每个 object 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 bart 绑定一个 name 属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

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，score 等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

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：

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.name
'Bart Simpson'
>>> bart.score
98
```

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、可变参数和关键字参数。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 和 score 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成绩：

```
>>> def print_score(std):
... print '%s: %s' % (std.name, std.score)
...
>>> print_score(bart)
Bart Simpson: 98
```

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, score):
        self.name = name
        self.score = score

    def print_score(self):
        print '%s: %s' % (self.name, self.score)
```

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：

```
>>> bart.print_score()
Bart Simpson: 98
```

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 score，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_grade：

```
class Student(object):
    ...

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

同样的，get_grade 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：

```
>>> bart.get_grade()
'A'
```

小结类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都不相同；通过在实例变量上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。和静态语言不同，Python 允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同：

```
>>> bart = Student('Bart Simpson', 98)
>>> lisa=Student('Lisa Simpson', 77)
>>> bart.age = 8
>>> bart.age
8
>>> lisa.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'age'
```

3.3.2 访问限制

重点：1 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 ___ 2 使用 get set 方法而不是直接用直接用 public 的原因：bart.score = 59 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：3 变量名类似 ___xxx___ 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___、___score___ 这样的变量名。4 使用 _name 的变量含义是：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（约定俗成）5 ___name___ 变量也可以访问，通过 _Student___name，但请不要这么干!!!

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、score 属性：

```
>>> bart = Student('Bart Simpson', 98)
>>> bart.score
98
>>> bart.score = 59
>>> bart.score
59
```

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：

```
class Student(object):
```

```
2
3     def __init__(self, name, score):
4         self.__name = name
5         self.__score = score
6
7     def print_score(self):
8         print '%s: %s' % (self.__name, self.__score)
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.__name 和实例变量.__score 了：

```
1 >>> bart = Student('Bart Simpson', 98)
2 >>> bart.__name
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 AttributeError: 'Student' object has no attribute '__name'
```

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 和 score 怎么办？可以给 Student 类增加 get_name 和 get_score 这样的方法：

```
1 class Student(object):
2     ...
3
4     def get_name(self):
5         return self.__name
6
7     def get_score(self):
8         return self.__score
```

如果又要允许外部代码修改 score 怎么办？可以给 Student 类增加 set_score 方法：

```
1 class Student(object):
2     ...
3
4     def set_score(self, score):
5         self.__score = score
```

你也许会问，原先那种直接通过 bart.score = 59 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：

```
1 class Student(object):
2     ...
3
4     def set_score(self, score):
5         if 0 <= score <= 100:
6             self.__score = score
7         else:
8             raise ValueError('bad score')
```

需要注意的是，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__、__score__ 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：

```
>>> bart._Student__name
'Bart Simpson'
```

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。

3.3.3 继承和多态

重点：1 面向对象的开闭原则，对扩展开放：允许新增 Animal 子类；对修改封闭：不需要修改依赖 Animal 类型的 run_twice() 等函数。

在 Python 中，类的基类只是简单地列在类名后面的小括号里。Python 支持多重继承。在类名后面的小括号中，你可以列出许多你想要的类名，以逗号分隔。比如:

```
class FileInfo(dict):#UserDict
```

如果一个 Python 函数，类方法，或属性的名字以两个下划线开始 (但不是结束)，它是私有的；其它所有的都是公有的。Python 没有类方法保护的概念 (只能用于它们自己的类和子类中)。类方法或者是私有 (只能在它们自己的类中使用) 或者是公有 (任何地方都可使用)。

在 Python 中，所有的专用方法 (像 __setitem__) 和内置属性 (像 __doc__) 遵守一个标准的命名习惯：开始和结束都有两个下划线。不要对你自己的方法和属性用这种方法命名；到最后，它只会把你 (或其它人) 搞乱。

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

```
class Animal(object):
    def run(self):
        print 'Animal is running...'
```

当我们需要编写 Dog 和 Cat 类时，就可以直接从 Animal 类继承：

```
class Dog(Animal):
    pass

```

```
class Cat(Animal):
    pass
```

对于 Dog 来说，Animal 就是它的父类，对于 Animal 来说，Dog 就是它的子类。Cat 和 Dog 类似。继承有什么好处？最大的好处是子类获得了父类的全部功能。由于 Animial 实现了 run() 方法，因此，Dog 和 Cat 作为它的子类，什么事也没干，就自动拥有了 run() 方法：

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：Animal is running... Animal is running...

当然，也可以对子类增加一些方法，比如 Dog 类：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'
    def eat(self):
        print 'Eating meat...'
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是 Dog 还是 Cat，它们 run() 的时候，显示的都是 Animal is running...，符合逻辑的做法是分别显示 Dog is running... 和 Cat is running...，因此，对 Dog 和 Cat 类改进如下：

```
class Dog(Animal):
    def run(self):
        print 'Dog is running...'

class Cat(Animal):
    def run(self):
        print 'Cat is running...'
```

再次运行，结果如下：Dog is running... Cat is running...

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：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用 isinstance() 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来 a、b、c 确实对应着 list、Animal、Dog 这 3 种类型。但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来 c 不仅仅是 Dog，c 还是 Animal！不过仔细想想，这是有道理的，因为 Dog 是从 Animal 继承下来的，当我们创建了一个 Dog 的实例 c 时，我们认为 c 的数据类型是 Dog 没错，但 c 同时也是 Animal 也没错，Dog 本来就是 Animal 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：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

Dog 可以看成 Animal，但 Animal 不可以看成 Dog。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 Animal 类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入 Animal 的实例时，run_twice() 就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入 Dog 的实例时，run_twice() 就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入 Cat 的实例时，run_twice() 就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

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

```
class Tortoise(Animal):
    def run(self):
        print 'Tortoise is running slowly...'
```

当我们调用 run_twice() 时，传入 Tortoise 的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个 Animal 的子类，不必对 run_twice() 做任何修改，实际上，任何依赖 Animal 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 Dog、Cat、Tortoise……时，我们只需要接收 Animal 类型就可以了，因为 Dog、Cat、Tortoise……都是 Animal 类型，然后，按照 Animal 类型进行操作即可。由于 Animal 类型有 run() 方法，因此，传入的任意类型，只要是 Animal 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 Animal 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 Animal、Dog、Cat 还是 Tortoise 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 Animal 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 Animal 子类；对修改封闭：不需要修改依赖 Animal 类型的 run_twice() 等函数。继承还可以一级一级地继承下来，就好比从祖父到爷爷、再到爸爸这样的关系。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：

小结继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写；有了继承，才能有多态。在调用类实例方法的时候，尽量把变量视作父类类型，这样，所有子类类型都可以正常被接收；旧的方式定义 Python 类允许不从 object 类继承，但这种编程方式已经严重不推荐使用。任何时候，如果没有合适的类可以继承，就继承自 object 类。

3.3.4 获取对象信息

重点：1 首先，我们来判断对象类型，使用 type() 函数：2 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 3 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：4 还有其他的内置函数

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？使用 type() 首先，我们来判断对象类型，使用 type() 函数：基本类型都可以用 type() 判断：

```
>>> type(123)
```

```
<type 'int'>
>>> type('str')
<type 'str'>
>>> type(None)
<type 'NoneType'>
```

如果一个变量指向函数或者类，也可以用 type() 判断：

```
>>> type(abs)
<type 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

但是 type() 函数返回的是什么类型呢？它返回 type 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：

```
>>> type(123)==type(456)
True
>>> type('abc')==type('123')
True
>>> type('abc')==type(123)
False
```

但是这种写法太麻烦，Python 把每种 type 类型都定义好了常量，放在 types 模块里，使用之前，需要先导入：

```
>>> import types
>>> type('abc')==types.StringType
True
>>> type(u'abc')==types.UnicodeType
True
>>> type([])==types.ListType
True
>>> type(str)==types.TypeType
True
```

最后注意到有一种类型就叫 TypeType，所有类型本身的类型就是 TypeType，比如：

```
>>> type(int)==type(str)==types.TypeType
True
```

使用 isinstance() 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。我们回顾上次的例子，如果继承关系是：object -> Animal -> Dog -> Husky

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 = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

没有问题，因为 h 变量指向的就是 Husky 对象。再判断：

```
>>> isinstance(h, Dog)
True
```

h 虽然自身是 Husky 类型，但由于 Husky 是从 Dog 继承下来的，所以，h 也还是 Dog 类型。换句话说，isinstance() 判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。因此，我们可以确信，h 还是 Animal 类型：

```
>>> isinstance(h, Animal)
True
```

同理，实际类型是 Dog 的 d 也是 Animal 类型：

```
>>> isinstance(d, Dog) and isinstance(d, Animal)
True
```

但是，d 不是 Husky 类型：

```
>>> isinstance(d, Husky)
False
```

能用 type() 判断的基本类型也可以用 isinstance() 判断：

```
>>> isinstance('a', str)
True
>>> isinstance(u'a', unicode)
True
>>> isinstance('a', unicode)
False
```

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 str 或者 unicode：

```
>>> isinstance('a', (str, unicode))
True
>>> isinstance(u'a', (str, unicode))
True
```

由于 str 和 unicode 都是从 basestring 继承下来的，所以，还可以把上面的代码简化为：

```
>>> isinstance(u'a', basestring)
True
```

使用 dir() 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：

```
>>> dir('ABC')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '
    __getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '
    __le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '
    __repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '
    _formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', '
    endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
    'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', '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：

```
>>> len('ABC')
3
>>> 'ABC'.___len___()
3
```

我们自己写的类，如果也想用 len(myObj) 的话，就自己写一个 ___len___() 方法：

```
>>> class MyObject(object):
... def ___len___(self):
... return 100
...
>>> obj = MyObject()
>>> len(obj)
100
```

剩下的都是普通属性或方法，比如 lower() 返回小写的字符串：

```
>>> 'ABC'.lower()
'abc'
```

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 MyObject(object):
... def ___init___(self):
... self.x = 9
... def power(self):
... return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗?
True
```

```
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗?
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗?
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
19
```

如果试图获取不存在的属性，会抛出 AttributeError 的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```

可以传入一个 default 参数，如果属性不存在，就返回默认值：

```
>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404
404
```

也可以获得对象的方法：

```
>>> hasattr(obj, 'power') # 有属性'power'吗?
True
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <___main___.MyObject object at 0x108ca35d0>>
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn
>>> fn # fn指向obj.power
<bound method MyObject.power of <___main___.MyObject object at 0x108ca35d0>>
>>> fn() # 调用fn()与调用obj.power()是一样的
81
```

小结通过内置的一系列函数，我们可以对任意一个 Python 对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：sum = obj.x + obj.y

就不要写：

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

假设我们希望从文件流 fp 中读取图像，我们首先要判断该 fp 对象是否存在 read 方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr() 就派上了用场。请注意，在 Python 这类动态语言中，有 read() 方法，不代表该 fp 对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要 read() 方法返回的是有效的图像数据，就不影响读取图像的功能。

3.4 IO 编程

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 就是数据从内存流到外面（磁盘、网络）去，Output Stream 就是数据从外面流进来。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。

3.4.1 目录

os.path 模块有几个操作文件和目录的函数。这里，我们看看如何操作路径名和列出一个目录的内容。os.path 是一个模块的引用；使用哪一个模块要看你正运行在哪种平台上。就像 getpass 通过将 getpass 设置为一个与平台相关的函数从而封装了平台之间的不同。os 通过设置 path 封装不同的相关平台模块。os.path 的 join 函数把一个或多个部分路径名连接成一个路径名。在这个简单的例子中，它只是将字符串进行连接。(请注意在 Windows 下处理路径名是一个麻烦的事，因为反斜线字符必须被转义。) 在这个几乎没有价值的例子中，在将路径名加到文件名上之前，join 将在路径名后添加额外的反斜线。当发现这一点时我高兴极了，因为当用一种新的语言创建我自己的工具包时，addSlashIfNecessary 总是我必须要写的那些愚蠢的小函数之一。在 Python 中不要写这样的愚蠢的小函数，聪明的人已经为你考虑到了。expanduser 将对使用 来表示当前用户根目录的路径名进行扩展。在任何平台上，只要用户拥有一个根目录，它就会有效，像 Windows、UNIX 和 Mac OS X，但在 Mac OS 上无效。

split 函数对一个全路径名进行分割，返回一个包含路径和文件名的 tuple。还记得我说过你可以使用多变量赋值从一个函数返回多个值吗？对，split 就是这样一个函数。os.path 也包含了一个 splitext 函数，可以用来对文件名进行分割，并且返回一个包含了文件名和文件扩展名的 tuple。我们使用相同的技术来将它们赋值给独立的变量。

listdir 函数接收一个路径名，并返回那个目录的内容的 list。listdir 同时返回文件和文件夹，并不指出哪个是文件，哪个是文件夹。你可以使用过滤列表和 os.path 模块的 isfile 函数，从文件夹中将文件分离出来。isfile 接收一个路径名，如果路径表示一个文件，则返回 1，否则为 0。在这里，我们使用 os.path.join 来确保得到一个全路径名，但 isfile 对部分路径 (相对于当前目录) 也是有效的。你可以使用 os.getcwd() 来得到当前目录。os.path 还有一个 isdir 函数，当路径表示一个目录，则返回 1，否则为 0。你可以使用它来得到一个目录下的子目录列表。

3.4.2 文件读写

Python 有一个内置函数，open，用来打开在磁盘上的文件。open 返回一个文件对象，它拥有一些方法和属性，可以得到被打开文件的信息，以及对被打开文件进行操作。

open 方法可以接收三个参数：文件名、模式和缓冲区参数。只有第一个参数 (文件名) 是必须的；其它两个是可选的。如果没有指定，文件以文本方式打开。这里我们以二进制方式打开文件进行读取。(print open.___doc___ 会给出所有可能模式的很好的解释。) open 函数返回一个对象 (到现在为止，这一点应该不会使你感到吃惊)。一个文件对象有几个有用的属性。文件对象的 mode 属性告诉你文件以何种模式被打开。文件对象的 name 属性告诉你文件对象所打开的文件名。

一个文件对象维护它所打开文件的状态。文件对象的 tell 方法告诉你在被打开文件中的当前位置。因为我们还没有对这个文件做任何事，当前位置为 0，它是文件的起始处。文件对象的 seek 方法在被打开文件中移动到另一个位置。第二个参数指出第一个参数是什么意思：0 表示移动到一个绝对位置 (从文件起始处算起)，1 表示移到一个相对位置 (从当前位置算起)，还有 2 表示相对于文件尾的位置。因为我们搜索的 MP3 标记保存在文件的末尾，我们使用 2 并且告诉文件对象从文件尾移动到 128 read 方法从被打开文件中读取指定个数的字节，并且返回含有读取数据的字符串。可选参数指定了读取的最大字节数。如果没有指定参数，read 将读到文件末尾。(我们本可以在这里简单地说 read() ，因为我们确切地知道在文件的何处，事实上，我们读的是最后 128 个字节。) 读出的数据赋给变量 tagData，并且当前的位置根据所读的字节数作了修改。(参考 dive in pythobn 第 6 章)

文件对象的 closed 属性表示对象是打开还是关闭了文件。在本例中，文件仍然打开着 (closed 是 False)。为了关闭文件，调用文件对象的 close 方法。这样就释放掉你加在文件上的锁 (如果有的话)，刷新被缓冲的系统还未写入的输出 (如果有的话)，并且释放系统资源。文件被关闭了，但这并不意味着文件对象不再存在。变量 f 将继续存在，直到它超出作用域或被手工删除。然而，一旦文件被关闭，操作它的方法就没有一个能使用；它们都会引发异常。对一个文件已经关闭的文件对象调用 close 不会引发异常，它静静地失败。

正如你所期待的，你也能用与读取文件同样的方式写入文件。有两种基本的文件模式：• 追加 (Append) 模式将数据追加到文件尾。• 写入 (write) 模式将覆盖文件的原有内容。如果文件还不存在，任意一种模式都将自动创建文件，因此从来不需要任何复杂的逻辑：“如果 log 文件还不存在，将创建一个新的空文件，正因为如此，你可以第一次就打开它”。打开文件并开始写就可以了。

读写文件是最常见的 IO 操作。Python 内置了读写文件的函数，用法和 C 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：

```
1 >>> f = open('/Users/michael/test.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：

```
1 >>> f=open('/Users/michael/notfound.txt', 'r')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 IOError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'
```

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：

```
>>> f.read()
'Hello, world!'
```

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：

```
>>> f.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：

```
try:
    f = open('/path/to/file', 'r')
    print f.read()
finally:
    if f:
        f.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('/path/to/file', 'r') as f:
    print f.read()
```

这和前面的 try ... finally 是一样的，但是代码更佳简洁，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：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

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 ASCII 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：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

字符编码要读取非 ASCII 编码的文本文件，就必须以二进制模式打开，再解码。比如 GBK 编码的文件：

```
>>> f = open('/Users/michael/gbk.txt', 'rb')
>>> u = f.read().decode('gbk')
>>> u
u'\u6d4b\u8bd5'
>>> print u
测试
```

如果每次都这么手动转换编码嫌麻烦（写程序怕麻烦是好事，不怕麻烦就会写出又长又难懂又没法维护的代码），Python 还提供了一个 codecs 模块帮我们在读文件时自动转换编码，直接读出 unicode：

```
import codecs
with codecs.open('/Users/michael/gbk.txt', 'r', 'gbk') as f:
    f.read() # u'\u6d4b\u8bd5'
```

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：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.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('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请效仿 codecs 的示例，写入 unicode，由 codecs 自动转换成指定编码。小结在 Python 中，文件读写是通过 open() 函数打开的文件对象完成的。使用 with 语句操作文件 IO 是个好习惯。

3.4.3 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='Bob', age=20, score=88)
```

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'Bob'。我们把变量存储到磁盘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，flattening 等等，都是一个意思。反过来，

从磁盘把变量内容重新读到内存里称之为反序列化，即 unpickling。Python 提供两个模块来实现序列化：cPickle 和 pickle。这两个模块功能是一样的，区别在于 cPickle 是 C 语言写的，速度快，pickle 是纯 Python 写的，速度慢，跟 cStringIO 和 StringIO 一个道理。用的时候，先尝试导入 cPickle，如果失败，再导入 pickle：

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

首先，我们尝试把一个对象序列化并写入文件：

```
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
"(dp0\nS'age'\np1\nI20\nsS'score'\np2\nI88\nsS'name'\np3\nS'Bob'\np4\ns."
```

pickle.dumps() 方法把任意对象序列化成一个 str，然后，就可以把这个 str 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：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.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 str，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：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。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。

Chapter 4

基于贝叶斯原理的 python 建模实践

参考: 贝叶斯思维-统计建模的 python 学习法

4.1 贝叶斯原理

条件概率联合概率全概率公式贝叶斯定理先验、后验、似然度、标准化常量

曲奇饼问题

假设有两个碗，碗 1 包含 30 个香草曲奇饼和 10 个巧克力曲奇饼，另一个碗 2 则包含两种饼干 20 个。假设随机从两个碗中随机挑选一块饼，得到一块香草曲奇饼，问是从碗 1 取到的概率是多少？

问题描述:

$$
\begin{aligned}
&P(\text{饼是从碗 1 取到} \mid \text{取到一块饼是香草曲奇饼}) \\
&= \frac{P(\text{饼是从碗 1 取到 and 取到一块饼是香草曲奇饼})}{P(\text{取到一块饼是香草曲奇饼})} \\
&= \frac{P(\text{一块饼是从碗 1 取到})P(\text{取到这块饼是香草曲奇饼} \mid \text{一块饼是从碗 1 取到})}{\begin{array}{l}P(\text{一块饼是从碗 1 取到})P(\text{取到这块饼是香草曲奇饼} \mid \text{一块饼是从碗 1 取到}) \\ +P(\text{一块饼是从碗 2 取到})P(\text{取到这块饼是香草曲奇饼} \mid \text{一块饼是从碗 1 取到})\end{array}}
\end{aligned}
\tag{4.1}
$$

结果为:

$$
P = \frac{0.5 * 0.75}{0.5 * 0.75 + 0.5 * 0.5} = 0.6 \tag{4.2}
$$

M&M 豆问题

M&M 豆有各种颜色的巧克力豆，制造商 Mars 公司不时变更不同颜色巧克力豆之间比例。1995 年推出了新的比例，此前一袋 M&M 豆中，比例为:30% 褐色，20% 黄色，20% 红色，10% 绿色，10% 橙色，10% 黄褐色，之后变成了 24% 蓝色，20% 绿色，16% 橙色，14% 黄色，13% 红色，13% 褐色。假设有两袋 M&M 豆，一袋是 1994 年，一袋是 1996 年，随机

从两个袋子各取出一个 M&M 豆，两个都一个是黄色，一个是绿色，那么黄色豆来在 1994 年的袋子的概率是多少？

问题描述:

$$\begin{aligned}
&P(\text{黄色豆来自 1994} \mid \text{从 1994 和 1996 两个袋各取出 1 个豆，一个黄色一个绿色}) \\
&= \frac{P(\text{黄色豆来自 1994 and 从 1994 和 1996 两个袋各取出 1 个豆，一个黄色一个绿色})}{P(\text{从 1994 和 1996 两个袋各取出 1 个豆，一个黄色一个绿色})} \\
&= \frac{P(\text{从 1994 袋取出黄色豆})P(\text{从另一个 1996 袋取出绿色豆} \mid \text{从 1994 袋取出黄色豆})}{P(\text{从 1994 袋取出黄色豆})P(\text{从另一个 1996 袋取出绿色豆} \mid \text{从 1994 袋取出黄色豆}) \\
&\quad + P(\text{不是从 1994 袋取出黄色豆})P(\text{从 1994 袋取出绿色豆} \mid \text{不是从 1994 袋取出黄色豆})}
\end{aligned} \tag{4.3}$$

结果为:

$$P = \frac{0.2 * 0.2}{0.2 * 0.2 + 0.14 * 0.1} = 0.740741 \tag{4.4}$$

蒙蒂大厅问题

这是一个游戏节目，大厅中有 A、B、C 三个门，一个门后面有奖品，另两个后面没有，随机配置。游戏的目的是猜哪个门后面是奖品，猜对了就可以拿走。游戏者任选一个门，在打开选中的门前，为增加悬念，主持人会打开另两个中一个没有奖品的门来增加悬念，主持人给你一个选中，坚持最初的选择还是换到剩下的那个未打开的门，问坚持原来的选择中奖的概率？

问题描述: 这个问题与主持人开门的策略有关，如果是随机选择开空门是一种情况，另一种是主持人总是选择其中一个门，除非该门不空。

看第一种策略的情况，假设游戏者最初选择的是 A:

$$\begin{aligned}
&P(\text{A 门有奖品} \mid \text{主持人随机选择打开 B/C 门，且打开的空门是 B}) \\
&= \frac{P(\text{A 门有奖品 and 主持人随机选择打开 B/C 门，且打开的空门是 B})}{P(\text{主持人随机选择打开 B/C 门，且打开的空门是 B})} \\
&= \frac{P(\text{A 门有奖品})P(\text{主持人随机选择打开 B/C 门，且打开的空门是 B| A 门有奖品})}{P(\text{A 门有奖品})P(\text{主持人随机选择打开 B/C 门，且打开的空门是 B| A 门有奖品}) \\
&\quad + P(\text{A 门为空})P(\text{主持人随机选择打开 B/C 门，且打开的空门是 B| A 门为空})}
\end{aligned} \tag{4.5}$$

结果为:

$$P = \frac{\frac{1}{3} * \frac{1}{2}}{\frac{1}{3} * \frac{1}{2} + \frac{2}{3} * 1} = \frac{1}{3} \tag{4.6}$$

看第二种策略的情况，假设游戏者最初选择的是 A:

$$\begin{aligned}
&P(\text{A 门有奖品} \mid \text{主持总是打开 B 门除非 B 门有奖品，且打开的空门是 B}) \\
&= \frac{P(\text{A 门有奖品 and 主持总是打开 B 门除非 B 门有奖品，且打开的空门是 B})}{P(\text{主持总是打开 B 门除非 B 门有奖品，且打开的空门是 B})} \\
&= \frac{P(\text{A 门有奖品})P(\text{主持总是打开 B 门除非 B 门有奖品，且打开的空门是 B| A 门有奖品})}{P(\text{A 门有奖品})P(\text{主持总是打开 B 门除非 B 门有奖品，且打开的空门是 B| A 门有奖品}) \\
&\quad + P(\text{A 门为空})P(\text{主持总是打开 B 门除非 B 门有奖品，且打开的空门是 B| A 门为空})}
\end{aligned} \tag{4.7}$$

结果为:

$$P = \frac{\frac{1}{3} * 1}{\frac{1}{3} * 1 + \frac{2}{3} * \frac{1}{2}} = \frac{1}{2} \tag{4.8}$$

4.2 分布计算

概率的归一化，即处理使累加概率等于 1。

Chapter 5

利用字符串和正则表达式的文本处理

5.1 字符串基本用法

字符串为引号之间的字符集合，这里引号包括单引号、双引号，三引号（三个连续的单引号或双引号）。

字符串是字符的序列，因此可以用序列的各种方法，比如索引，切片，in，not in，重复 *，拼接 +，求长度 len 等。有时还可以利用列表的一些方法执行一些特殊的操作，比如颠倒字符串

```
1 >>> text="hello"
2 >>> lst1=list(text)
3 >>> lst1.reverse()
4 >>> text="".join(lst1)
5 >>> text
6 'olleh'
```

删除的字符串用函数，del s，其中 s 为要删除的字符串名

一个反斜线加一个单一字符可以表示一个特殊字符，通常是一个不可打印的字符。例如`\t`表示一个字符，该字符为制表符。

原始字符串的定义即在字符串的前面加一个 r，如定义字符串 s，`s=r'I love\tPython'`，s 等价于`'I love\\tPython'`，制表符`\t` 在所定义的原字符串中，它失去了它的特殊含义。

常见转义字符包括:

- `\0` NUL 空字符 Nul
- `\a` BEL 响铃字符
- `\b` BS 退格
- `\t` HT 横向制表符
- `\n` LF 换行
- `\v` VT 纵向制表符

- \f FF 换页
- \r CR 回车
- \e ESC 转义
- \" " 双引号
- \' ' 单引号
- \\ 反斜杠

成对定义的单引号中单引号要转义，成对定义的双引号中双引号要转义，成对的三引号中单引号双引号都不需要转义，成对定义的单引号中双引号不需要转义，成对定义的双引号中单引号不需要转义

另一种转义是将数值转义成字符，比如:"\x6d"

用函数也可以把数值直接变为字符串:

- str(obj) 将其他类型内容转换为字符串
- repr(obj) 将对象转换为描述其的规范字符串

比如:

```
>>> lst2=[1,2,3]
>>> repr(lst2)
'[1, 2, 3]'
>>> str(lst2)
'[1, 2, 3]'
```

而字符串转换为数值相关函数包括:

- int(obj) 将字符串或整数转换为浮点数
- float(obj) 将字符串或整数转换为浮点数

例如:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

"""
测试-正则表达式
"""

import os
import sys
import subprocess
import re

def regtest():
    #eg from the quick python book
    print("\155") #8进制
    print("\012")
    print("\x6d") #16进制
    print("\x0a")
```

```
19      print("\u00e1") #UNICODE
20      print("\N{LATIN␣SMALL␣LETTER␣A}")
21      print("\u4756")
22
23  def transtest():
24      print(int("1000"))
25      print(int("1000",2))
26      print(int("1000",8))
27      print(int("1000",16))
28      print(float("1000"))
29
30  def main():
31      print("测试程序!")
32      print(os.getcwd())#路径问题
33      regtest()
34      transtest()
35      pass
36
37  if ___name___ == "__main__":
38      main()
```

字符串的格式化有两种方式:

1 是利用 format 方法

2 是利用% 生成

比如:

```
1   #!/usr/bin/env python3
2   #_*_coding: utf-8 _*_
3
4   """
5   测试-正则表达式
6   """
7
8   import os
9   import sys
10  import subprocess
11  import re
12
13  def regtest():
14      #eg from the quick python book
15      print("the␣{0}␣is␣{1}".format("书","新华字典"))
16      print("the␣{{书}}␣is␣{0}".format("新华字典"))
17      print("the␣{book}␣is␣{bookname}".format(book="书",bookname="新华字典"))
18      print("the␣{0}␣is␣{bookname}".format("书",bookname="新华字典"))
19      print("the␣{0}␣is␣{bookname[1]}".format("书",bookname=["新华字典","现代汉语词典"]))
20      print("the␣{0:10}␣is␣{1}".format("书","新华字典"))#使用填充格式，10是字符数
21      print("the␣{0:>10}␣is␣{1}".format("书","新华字典"))
22      print("the␣{0:&>10}␣is␣{1}".format("书","新华字典"))
23      print("the␣{0:*>10}␣is␣{1}".format("书","新华字典"))
```

```
24
25      print("the␣%s␣is␣%s"%("书","新华字典"))
26      print("%.2f␣-␣%.4f"%(3.14159,3.14159))
27      print("%(pi).2f␣-␣%(pi).4f" % {"pi":3.14159})
28
29  def main():
30      print("测试程序!")
31      print(os.getcwd())#路径问题
32      regtest()
33      pass
34
35  if __name__ == "__main__":
36      main()
```

字符串常用函数包括:

- str.find(sub[, start[, end]]) 判断 sub 是否在 str 中，如果 start，end 指定，则返回 str 中指定范围内 sub 出现的位置。在则返回在整个 str 中开始的索引值，不在则返回-1。
- str.rfind(sub[, start[, end]]) 与 find() 函数功能类似，只不过是从 str 右边开始查找。
- str.index(sub[, start[, end]]) 同 find() 函数功能相同，只不过当没有查找到 sub 时会抛出 ValueError 异常。
- str.rindex(sub[, start[, end]]) 与 index() 函数功能类似，只不过是从 str 右边开始查找。
- str.count(sub[, start[, end]]) 返回 sub 在 str 里面出现的次数，如果 start，end 指定，则返回 str 中指定范围内 sub 出现的次数。
- str.isalnum() 如果 str 至少含有一个字符，并且所有的字符都是字母或数字则返回 True，否则返回 False。(空格即不是字母也不是数字)
- str.isalpha()：判断是不是都是字母字符
- str.isdigit()：判断是不是都是数字
- str.islower()：判断是不是都是小写
- str.isspace()：判断是不是都是英文空格
- str.istitle()：判断是不是都是标题化文本
- str.isupper()：判断是不是都是大写
- str.startswith(prefix[, start[, end]]) 判断字符串是否以 prefix 开始（prefix 在这里可以为字符串或一个元组，为元组时则表示以任何一个元组中的字符串开始），如果 start，end 指定，则判断 str 指定范围内的子串是否以 prefix 开始。如果是返回 True，否则返回 False。
- str.endswith(suffix[, start[, end]]) 判断字符串是否以 suffix 结束，如果 start，end 指定，则返回 str 中指定范围内 str 子串是否以 suffix 为结尾。如果是返回 True，否则返回 False。
- str.replace(old, new[, count]) 将 str 中的 old 字符串替换为 new 字符串，并将替换后的

新字符串返回，如果 count 指定，则只替换前 count 个字符串。

- str.translate() 和 str.maketrans() 可以同时替换多个字符，比如:

```
>>> str1="afjalfa~x^fjaf(afalfaf%dfaljflajfl)"
>>> str1.translate(str.maketrans("~^()","!@[]"))
'afjalfa!x@fjaf[afalfaf%dfaljflajfl]'
```

- str.join(seq) 以 str 作为分隔符，将序列 seq 中所有元素（必须为字符串或转换为字符串）合并为一个新的字符串。
- str.split([sep[, maxsplit]]) 以 sep 字符串为分隔符对 str 进行分割得到一个字符串列表，如果 maxsplit 指定，则仅分割 maxsplit 次（这样，列表中最多有 maxsplit+1 个元素），如果没有指定 maxsplit，则全部分割。如果 sep 指定，则多个连续的 sep 字符串不会被认为是一个 sep 字符串，分割结果也会包含相应的空串。指定 sep 情况下分割一个空的字符串会得到 ['']。如果 sep 没有指定，多个连续的空格会被认为是一个分隔符，返回的字符串列表中也不含有空串。sep 省略或为 None 时分割一个空串或只含有英文空格的字符串会得到 []
- str.rsplit([sep[, maxsplit]]) 与 split() 函数功能类似，只不过是从 maxsplit 指定时，从右边开始分割 maxsplit 次。
- str.splitlines([keepends]) 返回一个列表，列表中每一个元素为字符串 str 的一行（用采用换行符分隔），元素可能为空。如果 keepends 没有指定或指定为假，则返回的列表元素中不包含换行符，如果指定，且不为假，则每个元素末尾都包含换行符。
- str.center(width[, fillchar]) 返回一个长为 width 的新字符串，在新字符中，原字符串居中，将 fillchar 指定的符号（默认为空格）填充至其前后。
- str.ljust(width[, fillchar]) 返回一个原字符串左对齐，并用指定符号 fillchar （默认为英文空格）填充至长度为 width 的新字符串。如果 width<len(str)，则返回 str。
- str.rjust(width[, fillchar]) 返回一个原字符串右对齐，并用指定符号 fillchar （默认为英文空格）填充至长度为 width 的新字符串。如果 width<len(str)，则返回 str。
- str.strip([chars]) 返回一个新的字符串，字符串的首尾中连续含有 chars 字符串内字符的字符被删除。如果不指定 chars，则默认删除 str 的首尾连续英文空格。

```
>>> "www.python.org".strip("w")
'.python.org'
>>> "www.python.org".strip("gor")
'www.python.'
>>> "www.python.org".strip(".gor")
'www.python'
>>> "www.python.org".strip(".ngor")
'www.pyth'
>>> "www.python.org".strip("w.ngor")
'pyth'
>>>
```

- str.lstrip([chars]) 与 strip() 函数功能类似，只删除 str 右侧连续包含 chars 字符串指定的字符。chars 不指定时，默认为英文空格。
- str.rstrip([chars]) 与 strip() 函数功能类似，只删除 str 右侧连续包含 chars 字符串指定的字符。chars 不指定时，默认为英文空格。
- str.title() 返回标题化的字符串：字符串内所有单词首字母大写，其余字母小写。如果单词以没有大小写变化的字符开始，则忽略这些字符，而将单词的第一个具有大小写变化的字符大写。整个单词都没有字符有大小写变化，则不改变这个单词。
- str.capitalize() 将字符串首字符大写，并返回新的首字符大写后的字符串。
- str.swapcase() 返回一个新的字符串，字符串中大写字符改为小写，小写字符改为大写。
- str.upper() 返回一个 str 中所有字符均大写的新字符串。
- str.lower() 返回一个 str 中所有字符均小写的新字符串。
- str.zfill(width) 返回长度为 width 的新字符串，str 位于新字符串右端，前面填充 0。如果 str 长度大于 width，则 str 会被返回。一般用于数字字符串。

5.2 元字符及其作用

正则表达式的元字符及其作用

```
1 . ^ $ * + ? { [ ] \ | ( )
```

元字符是”[” 和”]”。它们常用来指定一个字符类别，所谓字符类别就是你想匹配的一个字符集。字符可以单个列出，也可以用“-”号分隔的两个给定字符来表示一个字符区间。元字符在类别里并不起作用。补集来匹配不在区间范围内的字符。其做法是把”” 作为类别的首个字符；其它地方的”” 只会简单匹配”” 字符本身。

反斜杠” 是¨ 转义字符，做为 Python 中的字符串字母，反斜杠后面可以加不同的字符以表示不同特殊意义

元字符. 匹配除了换行字符外的任何字符，在 alternate 模式（re.DOTALL）下它甚至可以匹配换行。”.” 通常被用于你想匹配“任何字符”的地方。

其它模式包括:

DOTALL, S 使. 匹配包括换行在内的所有字符

IGNORECASE, I 使匹配对大小写不敏感

LOCALE, L 做本地化识别（locale-aware）匹配

MULTILINE, M 多行匹配，影响 和ˆ $

VERBOSE, X 能够使用 REs 的 verbose 状态，使之被组织得更清晰易懂

```
1 \d 匹配任何十进制数； 它相当于类 [0−9]。
2 \D 匹配任何非数字字符； 它相当于类 [^0−9]。
3 \s 匹配任何空白字符； 它相当于类 [ \t\n\r\f\v]。
4 \S 匹配任何非空白字符； 它相当于类 [^ \t\n\r\f\v]。
```

\w 匹配任何字母数字字符； 它相当于类 [a−zA−Z0−9_]。
\W 匹配任何非字母数字字符； 它相当于类 [^a−zA−Z0−9_]。
\b 是一个在单词边界定位当前位置的界定符（ assertions）。这是个零宽界定符（ zero−width assertions） 只用以匹配单词的词首和词尾。 单词被定义为
一个字母数字序列， 因此词尾就是用空白符或非字母数字符来标示的。
\B 另一个零宽界定符（ zero−width assertions）， 它正好同 \b 相反， 只在当前位置不在单词边界时匹配。
\A 只匹配字符串首。 当不在 MULTILINE 模式， \A 和 ^ 实际上是一样的。 然而， 在 MULTILINE 模式里
它们是不同的； \A 只是匹配字符串首， 而 ^ 还可以匹配在换行符之后字符串的任何位置。
\Z 只匹配字符串尾。

正则表达式第一件能做的事是能够匹配不定长的字符集，而这是其它能作用在字符串上的方法所不能做到的。另一个功能则是可以指定正则表达式的一部分的重复次数。

重复功能的元字符是 *。* 并不匹配字母字符"*"；相反，它指定前一个字符可以被匹配零次或更多次，而不是只有一次。* 这样地重复是“贪婪的”；当重复一个 RE 时，匹配引擎会试着重复尽可能多的次数。如果模式的后面部分没有被匹配，匹配引擎将退回并再次尝试更小的重复。

另一个重复元字符是 +，表示匹配一或更多次。请注意 * 和 + 之间的不同；*匹配零或更多次，所以可以根本就不出现，而 + 则要求至少出现一次。

问号? 匹配一次或零次；可以认为它用于标识某事物是可选的。它也有限制贪婪的作用。

最复杂的重复限定符是 m,n，其中 m 和 n 是十进制整数。该限定符的意思是至少有 m 个重复，至多到 n 个重复。如果忽略 m 或 n；因为会为缺失的值假设一个合理的值。忽略 m 会认为下边界是 0，而忽略 n 的结果将是上边界为无穷大。0, 等同于 *，1, 等同于 +，而 0,1 则与? 相同。如果可以的话，最好使用 *，+，或?。很简单因为它们更短也更容易懂。

| 可选项，或者"or" 操作符。

匹^ 配行首。除非设置 MULTILINE 标志，它只是匹配字符串的开始。在 MULTILINE 模式里，它也可以直接匹配字符串中的每个换行。

$ 匹配行尾，行尾被定义为要么是字符串尾，要么是一个换行字符后面的任何位置。

组是通过"(" 和")" 元字符来标识的。

组用"(" 和")" 来指定，并且得到它们匹配文本的开始和结尾索引；这就可以通过一个参数用 group()、start()、end() 和 span() 来进行检索。组是从 0 开始计数的。组 0 总是存在；它就是整个 RE，所以 MatchObject 的方法都把组 0 作为它们缺省的参数。

使用组捕获的内容的使用有三种方式:

5.3 使用正则表达式

re.compile() 也接受可选的标志参数,常用来实现不同的特殊功能和语法变更,比如 re.IGNORECASE 等。

re.match() 决定 RE 是否在字符串刚开始的位置匹配

表 5.1: 捕获内容的引用

引用环境	使用方式
正则表达式中	• (?P=quote) • \1
处理匹配对象 m 时	• m.group('quote') • m.end('quote') 等
在 sub 替换的 repl 参数即替换字符串中	• \g<quote> • \g<1> • \1

re.search() 扫描字符串，找到这个 RE 匹配的位置

(re.match() checks for a match only at the beginning of the string, while re.search() checks for a match anywhere in the string (this is what Perl does by default). Regular expressions beginning with ” can be used with search() to restrict the match at the beginning of the string。however that in MULTILINE mode match() only matches at the beginning of the string, whereas using search() with a regular expression beginning with ” will match at the beginning of each line.)

re.findall() 找到 RE 匹配的所有子串，并把它们作为一个列表返回

re.finditer() 找到 RE 匹配的所有子串，并把它们作为一个迭代器返回

如果没有匹配到的话，match() 和 search() 将返回 None。如果成功的话，就会返回一个 MatchObject 实例，其中有这次匹配的信息：它是从哪里开始和结束，它所匹配的子串等等

MatchObject 实例也有几个方法和属性，最重要的那些如下所示：

group() 返回被 RE 匹配的字符串

start() 返回匹配开始的位置

end() 返回匹配结束的位置

span() 返回一个元组包含匹配 (开始, 结束) 的位置

split() 将字符串在 RE 匹配的地方分片并生成一个列表，

sub() 找到 RE 匹配的所有子串，并将其用一个不同的字符串替换

subn() 与 sub() 相同，但返回新的字符串和替换次数

5.4 举例

应该熟悉下列技巧：

1 ^ 匹配字符串的开始。
2 $ 匹配字符串的结尾。
3 \b 匹配一个单词的边界。
4 \d 匹配任意数字。
5 \D 匹配任意非数字字符。

x? 匹配一个可选的 x 字符 (换言之，它匹配 1 次或者 0 次 x 字符)。
x* 匹配 0 次或者多次 x 字符。
x+ 匹配 1 次或者多次 x 字符。
x{n,m} 匹配 x 字符，至少 n 次，至多 m 次。
(a|b|c) 要么匹配 a，要么匹配 b，要么匹配 c。
(x) 一般情况下表示一个记忆组 (remembered group)。可以利用re.search 函数返回对象的 groups() 函数获取它的值。

举例 1:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

"""
测试-正则表达式
"""

import os
import sys
import subprocess
import re

def hexrepl( match ):
    "Return␣the␣hex␣string␣for␣a␣decimal␣number"
    value = int( match.group() )
    return hex(value)

def regtest():
    #第一种使用方式
    p = re.compile('ab*')
    print(p)
    #注意传递给compile的是正则字符串，但这是用python字符串表示的，因此需要用注意转义,比如要匹配\section，那么
        正则表达式应该是\\section，那么python字符串应该是\\\\section
    #当然可以利用原始字符串表示，比如r"\\section"
    m=p.match("")
    print(m)
    m=p.match("abcdefgabbb")
    print(m)
    m=p.search("abcdefgabbb")
    print(m)
    m=p.findall("abcdefgabbb")
    print(m)
    m=p.finditer("abcdefgabbb")
    print(m)

    p = re.compile('\d+')
    print(p.findall('12␣drummers␣drumming,␣11␣pipers␣piping,␣10␣lords␣a-leaping'))

    iterator = p.finditer('12␣drummers␣drumming,␣11␣...␣10␣...')
    for match in iterator:
        print(match.span())

```

```
42      #第二种
43      print(re.match(r'From\s+', 'Fromage amk'))
44      print(re.match(r'From+', 'Fromage amk'))
45      print(re.match(r'From', 'Fromage amk'))
46
47
48      #替换，使用捕获的匹配信息
49      #eg form A.M. Kuchling 的Python 正则表达式操作指南，http://www.amk.ca/python/howto/regex/
50      p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
51      print(p.sub(r'subsection{\1}','section{First}'))
52      print(p.sub(r'subsection{\g<1>}','section{First}'))
53      print(p.sub(r'subsection{\g<name>}','section{First}'))
54
55      p = re.compile(r'\d+')
56      print(p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.'))
57
58      #eg from Dive Into Python
59      phonePattern = re.compile(r'''
60  # don't match beginning of string, number can start anywhere
61  (\d{3}) # area code is 3 digits (e.g. '800')
62  \D* # optional separator is any number of non-digits
63  (\d{3}) # trunk is 3 digits (e.g. '555')
64  \D* # optional separator
65  (\d{4}) # rest of number is 4 digits (e.g. '1212')
66  \D* # optional separator
67  (\d*) # extension is optional and can be any number of digits
68  $ # end of string
69  ''', re.VERBOSE)
70      print(phonePattern.search('work 1-(800) 555.1212 #1234').groups())
71      print(phonePattern.search('800-555-1212').groups())
72      print(phonePattern.search('80055512121234').groups())
73      print(phonePattern.search('800.555.1212 x1234').groups())
74      print(phonePattern.search('800-555-1212').groups())
75      print(phonePattern.search('(800)5551212 x1234').groups())
76      print(phonePattern.search('800 555 1212 1234').groups())
77      print(phonePattern.search('800-555-1212-1234').groups())
78
79  def main():
80      print("测试程序!")
81      print(os.getcwd())#路径问题
82      regtest()
83      pass
84
85  if __name__ == "__main__":
86      main()
```

举例 2:

```
1   #!/usr/bin/env python3
2   #_*_coding: utf-8 _*_
3
```

```
4  """
5  测试一正则表达式
6  """
7
8  import os
9  import sys
10 import subprocess
11 import re
12
13 def regtest():
14     #eg from the quick python book
15     regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"
16             r"(?P<first>[-a-zA-Z]+)"
17             r"(␣(?P<middle>([-a-zA-Z]+)))?"#显然在r''原始字符串里面，直接用空格可以代替\s，比如这里等价于r"(\s
                   (?P<middle>([-a-zA-Z]+)))?"
18             r":␣(?P<phone>(\d\d\d-)?\d\d\d-\d\d\d\d)")
19     print(regexp)
20     f=open("info.dat","w")
21     f.write('''zhang,yi san: 081-8649\nwang,yi dao: 086-8451\n''')
22     f.close()
23     file = open("info.dat")
24     for line in file:
25         print(line.strip())
26         result = regexp.search(line)
27         print(result)
28         if result == None:
29             print("Oops,␣I␣don't␣think␣this␣is␣a␣record")
30         else:
31             lastname = result.group('last')
32             firstname = result.group('first')
33             middlename = result.group('middle')
34             if middlename == None:
35                 middlename = ""
36             phonenumber = result.group('phone')
37             print('Name:', firstname, middlename, lastname,'␣Number:', phonenumber)
38     file.close()
39
40     string = "If␣the␣the␣problem␣is␣textual,␣use␣the␣the␣re␣module"
41     pattern = r"the␣the"
42     regexp = re.compile(pattern)
43     print(regexp.sub("the", string))
44
45     int_string = "1␣2␣3␣4␣5"
46     def int_match_to_float(match_obj):
47         return(match_obj.group('num') + ".0\n")
48
49     pattern = r"(?P<num>[0-9]+)"
50     regexp = re.compile(pattern)
51     print(regexp.sub(int_match_to_float, int_string,3))#可以限制替换的数量
52
```

```
53      #注意空的正则表达式的问题
54      regexp=re.compile('x*')
55      print(regexp.sub('-','abc'))
56      print(re.sub('x*', '-', 'abc')) #Empty matches for the pattern are replaced only when not adjacent to a
            previous match, so sub('x*', '-', 'abc') returns '-a-b-c-'.
57      print(re.sub('', '-', "abc"))
58      print(re.sub('a', '-', 'abc'))
59      print(re.sub('^$', '-', 'abc'))
60      print(re.sub('^', '-', 'abc'))
61      print(re.sub('$', '-', 'abc'))
62
63      print(re.findall('x*','abc'))
64      print(re.findall('','abc'))
65      print(re.findall('a','abc'))
66      print(re.findall('^$', 'abc'))
67      print(re.findall('^', 'abc'))
68      print(re.findall('$', 'abc'))
69
70  def main():
71      print("测试程序!")
72      print(os.getcwd())#路径问题
73      regtest()
74      pass
75
76  if __name__ == "__main__":
77      main()
```

Chapter 6

利用 python 实现系统脚本实践

目的是实现: 运行脚本，系统文件的管理 (复制，删除)，文件内容的分析处理 (正则，文本，csv，xls，系统程序的调用，c/fortran 程序的调用。

6.1 在操作系统内操作文件和目录

举例:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

"""
测试系统内的一些操作
"""

import os
import shutil
import subprocess

print(os.linesep)#这些都是字符串
print(os.sep)
print(os.pathsep)
print(os.curdir)
print(os.pardir)

def deldir(path):#遍历删除指定路径包括其包含的所有文件或目录
    print("to del",path)
    if os.path.isfile(path):
        os.remove(path)
        print("del "+path)
    elif os.path.isdir(path):
        pathlist=os.listdir(path)
        if len(pathlist)!=0:
            for subpath in pathlist:
```

```
27                  pathnow=path+os.sep+subpath
28                  deldir(pathnow)
29          os.rmdir(path)
30          print("rm␣"+path)
31
32  def testdir():
33      os.mkdir("eg")
34      os.chdir(os.curdir+os.sep+"eg")
35      os.mkdir("subeg")
36      f=open("egfile.dat","w")
37      f.write("example␣file")
38      f.close()
39      print(os.getcwd())
40      os.chdir(os.pardir)
41      print(os.getcwd())
42      print(os.listdir())
43      pwd=os.getcwd()
44      for elem in os.listdir():
45          if os.path.isdir(elem):
46              os.chdir(pwd+os.sep+elem)
47              print(os.listdir())
48      os.chdir(os.pardir)
49
50  def testloopdir():#创建eg目录，写一个程序能写文件wtfile.py，创建5个子目录，将该文件复制到各个子目录下，并运行产
        生结果，然后将结果文件拷贝会到eg目录下，并合成为一个结果文件resall.dat
51      os.mkdir("eg")
52      os.chdir(os.getcwd()+os.sep+"eg")
53      f=open("wtfile.py","w")
54      f.write("#_*_coding:␣utf-8␣_*_\n")
55      f.write("import␣sys\n")
56      f.write("f=open('res.dat','w')\n")
57      f.write("f.write('file'+sys.argv[1])\n")
58      f.write("f.close()\n")
59      f.close()
60
61      for i in range(5):
62          path="subeg"+str(i)
63          os.mkdir(path)
64          os.chdir(os.getcwd()+os.sep+path)
65          shutil.copyfile(os.pardir+os.sep+"wtfile.py","wtfile.py")
66
67          # exec只能执行代码对象，因此对脚本输入参数是不可能的
68          # exec_code = compile(open("wtfile.py").read(),"",'exec')
69          # exec(exec_code)
70
71          subprocess.run("python␣wtfile.py␣"+str(i))
72          os.chdir(os.pardir)
73
74      f=open('resall.dat','w')
75      for i in range(5):
```

```
76              path="subeg"+str(i)
77              os.chdir(os.getcwd()+os.sep+path)
78              stra=open("res.dat").read()
79              f.write(stra+"\n")
80              os.chdir(os.pardir)
81          f.close()
82
83
84
85
86      def main():
87          print("测试程序!")
88          # testdir()
89          print(os.getcwd())#路径问题
90          if os.path.exists("eg"):
91              print(os.getcwd()+os.sep+"eg")
92              deldir(os.getcwd()+os.sep+"eg")
93          testloopdir()
94          pass
95
96
97      if __name__ == "__main__":
98          main()
99
100     #os模块
101
102     # linesep 用于在文件中分隔行的字符串
103     # sep 用来分隔文件路径名的字符串
104     # pathsep 用于分隔文件路径的字符串
105     # curdir 当前工作目录的字符串名称
106     # pardir (当前工作目录的)父目录字符串名称
107
108     # mkfifo()/mknod()a 创建命名管道/创建文件系统节点
109     # remove()/unlink() Delete file 删除文件
110     # rename()/renames()b 重命名文件
111     # *statc() 返回文件信息
112     # symlink() 创建符号链接
113     # utime() 更新时间戳
114     # tmpfile() 创建并打开('w+b')一个新的临时文件
115     # walk()a 生成一个目录树下的所有文件名
116     # 目录/文件夹
117     # chdir()/fchdir()a 改变当前工作目录/通过一个文件描述符改变当前工作目录
118     # chroot()d 改变当前进程的根目录
119     # listdir() 列出指定目录的文件
120     # getcwd()/getcwdu()a返回当前工作目录/功能相同, 但返回一个 Unicode 对象
121     # mkdir()/makedirs() 创建目录/创建多层目录
122     # rmdir()/removedirs() 删除目录/删除多层目录
123     # 访问/权限
124     # access() 检验权限模式
125     # chmod() 改变权限模式
```

```
126 # chown()/lchown()a 改变 owner 和 group ID/功能相同, 但不会跟踪链接
127 # umask() 设置默认权限模式
128 # 文件描述符操作
129 # open() 底层的操作系统 open (对于文件, 使用标准的内建 open() 函数)
130 # read()/write() 根据文件描述符读取/写入数据
131 # dup()/dup2() 复制文件描述符号/功能相同, 但是是复制到另一个文件描述符
132 # 设备号
133 # makedev()a 从 major 和 minor 设备号创建一个原始设备号
134 # major()a /minor()a 从原始设备号获得 major/minor 设备号
135
136 #os.path模块
137 # basename() 去掉目录路径, 返回文件名
138 # dirname() 去掉文件名, 返回目录路径
139 # join() 将分离的各部分组合成一个路径名
140 # split() 返回 (dirname(), basename()) 元组
141 # splitdrive() 返回 (drivename, pathname) 元组
142 # splitext() 返回 (filename, extension) 元组
143 # 信息
144 # getatime() 返回最近访问时间
145 # getctime() 返回文件创建时间
146 # getmtime() 返回最近文件修改时间
147 # getsize() 返回文件大小(以字节为单位)
148 # 查询
149 # exists() 指定路径(文件或目录)是否存在
150 # isabs() 指定路径是否为绝对路径
151 # isdir() 指定路径是否存在且为一个目录
152 # isfile() 指定路径是否存在且为一个文件
153 # islink() 指定路径是否存在且为一个符号链接
154 # ismount() 指定路径是否存在且为一个挂载点
155 # samefile() 两个路径名是否指向同个文件
```

6.2 在两层目录内编译 latex 文件

```
1  #!/usr/bin/env python3
2  #_*_coding: utf-8 _*_
3
4  """
5  两层目录下的循环编译
6  """
7
8  import os
9  import shutil
10 import subprocess
11 import sys
12
13 def compileall():
14     print(sys.platform)
15     if 'win' in sys.platform:
```

```
16          osflag=1
17      else:
18          osflag=0
19
20
21      path=os.getcwd()+os.sep+"exampleandimage"
22      os.chdir(path)
23      print("pwd=",os.getcwd())
24      for file in os.listdir():
25          if os.path.isfile(file):
26              fileext=os.path.splitext(file)[1]
27              filenam=os.path.splitext(file)[0]
28              if fileext==".tex":
29                  print(filenam,fileext)
30                  if osflag==1:
31                      subprocess.run("cmd␣/C␣del␣/q␣*.aux␣*.bbl␣*.blg␣*.log␣*.out␣*.toc␣*.bcf␣*.xml␣*.synctex␣
                            *.nlo␣*.nls␣*.bak␣*.ind␣*.idx␣*.ilg␣*.lof␣*.lot␣*.ent-x␣*.tmp␣*.ltx␣*.los␣*.lol␣*.
                            loc␣*.listing␣*.gz␣*.userbak␣*.nav␣*.snm␣*.vrb")
32                      subprocess.run("xelatex.exe␣--synctex=-1␣"+file)#注意python2中需要把执行程序和参数用列表来
                            表示，比如subprocess.Popen(["xelatex.exe","--synctex=-1",file])
33                      subprocess.run("biber.exe␣"+filenam)
34                      subprocess.run("xelatex.exe␣--synctex=-1␣"+file)
35                  else:
36                      subprocess.run("rm␣-r␣*.aux␣*.bbl␣*.blg␣*.log␣*.out␣*.toc␣*.bcf␣*.xml␣*.synctex␣*.nlo␣*.
                            nls␣*.bak␣*.ind␣*.idx␣*.ilg␣*.lof␣*.lot␣*.ent-x␣*.tmp␣*.ltx␣*.los␣*.lol␣*.loc␣*.
                            listing␣*.gz␣*.userbak␣*.nav␣*.snm␣*.vrb")
37                      subprocess.run("xelatex␣--synctex=-1␣"+file)
38                      subprocess.run("biber␣"+filenam)
39                      subprocess.run("xelatex␣--synctex=-1␣"+file)
40
41      if osflag==1:
42          subprocess.run("cmd␣/C␣del␣/q␣*.aux␣*.bbl␣*.blg␣*.log␣*.out␣*.toc␣*.bcf␣*.xml␣*.synctex␣*.nlo␣*.
                nls␣*.bak␣*.ind␣*.idx␣*.ilg␣*.lof␣*.lot␣*.ent-x␣*.tmp␣*.ltx␣*.los␣*.lol␣*.loc␣*.listing␣*.gz
                ␣*.userbak␣*.nav␣*.snm␣*.vrb")
43      else:
44          subprocess.run("rm␣-r␣*.aux␣*.bbl␣*.blg␣*.log␣*.out␣*.toc␣*.bcf␣*.xml␣*.synctex␣*.nlo␣*.nls␣*.bak␣
                *.ind␣*.idx␣*.ilg␣*.lof␣*.lot␣*.ent-x␣*.tmp␣*.ltx␣*.los␣*.lol␣*.loc␣*.listing␣*.gz␣*.userbak
                ␣*.nav␣*.snm␣*.vrb")
45      os.chdir(os.pardir)
46      print("pwd=",os.getcwd())
47      if osflag==1:
48          subprocess.run("xelatex.exe␣--synctex=-1␣biblatex-solution-to-latex-bibliography.tex")
49          subprocess.run("biber␣biblatex-solution-to-latex-bibliography")
50          subprocess.run("xelatex.exe␣--synctex=-1␣biblatex-solution-to-latex-bibliography.tex")
51          subprocess.run("xelatex.exe␣--synctex=-1␣biblatex-solution-to-latex-bibliography.tex")
52          subprocess.run("cmd␣/C␣del␣/q␣*.aux␣*.bbl␣*.blg␣*.log␣*.out␣*.toc␣*.bcf␣*.xml␣*.synctex␣*.nlo␣*.
                nls␣*.bak␣*.ind␣*.idx␣*.ilg␣*.lof␣*.lot␣*.ent-x␣*.tmp␣*.ltx␣*.los␣*.lol␣*.loc␣*.listing␣*.gz
                ␣*.userbak␣*.nav␣*.snm␣*.vrb")
53      else:
54          subprocess.run("xelatex␣--synctex=-1␣biblatex-solution-to-latex-bibliography.tex")
```

```
55          subprocess.run("biber␣biblatex-solution-to-latex-bibliography")
56          subprocess.run("xelatex␣--synctex=-1␣biblatex-solution-to-latex-bibliography.tex")
57          subprocess.run("xelatex␣--synctex=-1␣biblatex-solution-to-latex-bibliography.tex")
58          subprocess.run("rm␣-r␣*.aux␣*.bbl␣*.blg␣*.log␣*.out␣*.toc␣*.bcf␣*.xml␣*.synctex␣*.nlo␣*.nls␣*.bak␣
                *.ind␣*.idx␣*.ilg␣*.lof␣*.lot␣*.ent-x␣*.tmp␣*.ltx␣*.los␣*.lol␣*.loc␣*.listing␣*.gz␣*.userbak
                ␣*.nav␣*.snm␣*.vrb")
59
60
61
62  def main():
63      print("测试程序!")
64      print(os.getcwd())#路径问题
65      compileall()
66      pass
67
68
69  if ___name___ == "__main__":
70      main()
```

6.3 修改文本文件的内容

```
1   #!/usr/bin/env python3
2   #_*_coding: utf-8 _*_
3
4   """
5   测试对文件内容的修改
6   """
7
8   import os
9   import shutil
10  import subprocess
11
12  #首先从ctrlbak.dat中复制一个源文件ctrl.dat，然后修改ctrl.dat的文档内容，找到包含can的行修改参数为50，nit的行的参
        数改为10000，在42行后，每行前面加!符号,在攻角一行末尾加上\par字符
13  #先写到临时文件中，然后改名完成
14  def modifydatfile():
15      if os.path.exists("ctrlbak.dat"):
16          shutil.copyfile("ctrlbak.dat","ctrl.dat")
17          os.remove("ctrlbak.dat")
18      f=open("ctrl.dat")
19      g=open("temp.dat","w")
20      nline=0
21      for line in f:
22          nline+=1
23          if "can" in line:
24              eqidx=line.index("=")
25              linenew=line[0:eqidx+1]+"␣50\n"
26              g.write(linenew)
```

```
27          elif "nit" in line:
28              eqidx=line.index("=")
29              linenew=line[0:eqidx+1]+"␣10000\n"
30              g.write(linenew)
31          elif nline > 41 and line!="":
32              if "攻角" in line:
33                  linenew="!"+line[:-1]+r"\par"+"\n"
34                  g.write(linenew)
35              else:
36                  linenew="!"+line
37                  g.write(linenew)
38          else:
39              g.write(line)
40      g.close()
41      f.close()
42      os.rename("ctrl.dat","ctrlbak.dat")
43      os.rename("temp.dat","ctrl.dat")
44
45  def main():
46      print("测试程序!")
47      print(os.getcwd())#路径问题
48      modifydatfile()
49      pass
50
51  if __name__ == "__main__":
52      main()
```

6.4 设计一个 fortran 程序编译脚本

```
1   #!/usr/bin/env python3
2   #_*_coding: utf-8 _*_
3
4   """
5   测试--一个fortran编译脚本，有参数输入
6   """
7
8   import os
9   import sys
10  import subprocess
11
12  def compilefortrancode():
13      srcfile=""
14      ppn=""
15      try:
16          assert len(sys.argv)>2 #这里检查的还是比较简单的，只是检查了一下数量，其实还需要判断一下文件的扩展名，判
                断一下最后一个参数是不是数字
17      except AssertionError:
18          print("错误:输入参数不足，请重新输入!!!␣至少需要指定一个源代码文件和一个进程数")
```

```
19          sys.exit()
20      else:
21          srcfile="␣".join(sys.argv[1:-1]) #源文件
22          ppn=sys.argv[-1] #最后的数字
23          exefile=sys.argv[1][:-4] #第一个fortran文件名作为执行文件名
24          print("srcfile=",srcfile)
25          print("ppn␣␣␣␣=",ppn)
26          print("exefile=",exefile)
27
28      #subprocess.run函数似乎参数应该是一个字符串，而不是一个字符串列表，Popen 函数是不是这样需要进一步确认一下
29      #subprocess.run(["cmd /C","gfortran -o",exefile,srcfile,"-l fmpich2g -L D:\mingw\mpich2\lib -I D:\mingw\
            mpich2\include -O3 -ffree-line-length-0"])
30      subprocess.run("cmd␣/C␣gfortran␣-o"+exefile+"␣"+srcfile+"␣-l␣fmpich2g␣-L␣D:\mingw\mpich2\lib␣-I␣D:\
            mingw\mpich2\include␣-O3␣-ffree-line-length-0")
31
32      #直接执行没有任何问题，包括mpiexec也可以
33      if int(ppn) == 1:
34          subprocess.run(exefile)
35      else:
36          subprocess.run("mpiexec␣-n␣"+ppn+"␣"+exefile)
37
38  def main():
39      print("测试程序!")
40      print(os.getcwd())#路径问题
41      compilefortrancode()
42      pass
43
44  if ___name___ == "__main__":
45      main()
```

6.5 一个调用 ping 命令获取其结果的代码

利用 subprocess 模块:

```
1  #!/usr/bin/env python3
2  #_*_coding: utf-8 _*_
3
4  import subprocess
5
6  print(subprocess.getoutput('ping␣127.0.0.1'))
7  begin=101
8  end=200
9  while begin<end:
10     print(subprocess.getoutput('ping␣192.168.0.'+str(begin)))
11     begin+=1
```

Chapter 7

通过一个例子搞定 python 多线程和 socket 编程

一般来说，多线程的意义在于能够并发执行，也就是同时执行某个任务中分任务，把原来按顺序执行的任务同时执行，这种并发 (并行) 方式在数值计算中是很常见的。我以前做并行数值计算是在 CFD 中，利用 mpich+fortran 形成多进程并行的计算，当然还有其它的如 openmp，gpu 并行等方式，但主要采用这种方式。这种方式主要进程的并行计算。

通常任务进程是一个执行单元，而线程则是进程的一部分，一个进程可以由多个线程。根据我的理解这可能是在 windows 下的情况，在 unix 系统下，进程和线程从机制上并没有多大区别，可以认为是一个东西。在实现并发的时候，多个线程执行不同的子任务可以加速整个大任务的完成。这是一种并发的目的。还有一种则是服务的问题，多个线程可以同时提供多个相同的服务，而单线程则需要排队。

下面我们考虑一个简单的应用场景，假设有一台服务器，服务器可以同时连接多个用户，各用户可以向服务器发送信息并能从服务器得到反馈信息。我们知道向服务器发送消息可以利用 socket 的 tcp 或 udp 协议实现。其中 udp 发送信息不需要建立连接，因此多个用户同时发信息时就没有先后，也可能存在丢包等问题。而通过 tcp 发送信息则需要连接，但关键在于一个线程只能同时存在一个连接，因此实现 tcp 服务器同时连接多个用户的关键是要采用多线程。

7.1 threading 模块和多线程

因此我看首先看多线程的问题。在理解基本的线程概念后，我们可以利用 python 的线程模块来实现一些并发的执行，首先看下面一个例子 (来自:python 核心编程第 18 章):

```
1 #!/usr/bin/env python3
2 #_*_coding: utf-8 _*_
```

```
3
4  import threading
5  import random
6  from time import sleep, ctime
7
8  def fib(x):#计算斐波那契数列值的函数
9      sleep(0.005)
10     if x < 2: return 1
11     return (fib(x-2) + fib(x-1))
12
13 def fac(x):#计算阶乘函数
14     sleep(0.1)
15     if x < 2: return 1
16     return (x * fac(x-1))
17
18 def sum(x):#计算加和函数
19     sleep(0.1)
20     if x < 2: return 1
21     return (x + sum(x-1))
22
23 funcs = [fib, fac, sum]
24 n = 12
25
26 class MyThread(threading.Thread):#定义一个线程类MyThread
27     def __init__(self, func, args, name=''):#MyThread的构造函数
28         threading.Thread.__init__(self)
29         self.name = name
30         self.func = func
31         self.args = args
32         self.res = self.func(self.args)
33         self.gres=0
34     def getResult(self):#MyThread中的函数
35         return self.res
36     def taskexec(self):#MyThread中的函数
37         self.gres+=random.random()
38     def printres(self):#MyThread中的函数
39         print(self.gres)
40
41
42 def main():#脚本运行时的主测试函数
43     print('starting at:', ctime())
44     threads = []
45     nfuncs = range(len(funcs))
46
47     #单线程计算
48     print('*** SINGLE THREAD')
49     for i in nfuncs:
50         print('starting', funcs[i].__name__, 'at:',ctime())
51         print(funcs[i](n))
52         print(funcs[i].__name__, 'finished at:',ctime())
```

```
53
54      #多线程计算
55      print('***␣MUTIPLE␣THREAD')
56      for i in nfuncs:
57          t = MyThread(funcs[i], n,funcs[i].___name___)#注意构造MyThread线程时已经开始计算
58          threads.append(t)#主要到MyThread线程是一个对象
59
60      for i in nfuncs:
61          threads[i].start()#调用MyThread的活动，即run函数定义的内容，尽管这里没有定义run函数，但start还是要用，否
                则后面使用join等待是不行的。
62
63      for i in nfuncs:
64          threads[i].join()#Wait until the thread terminates，等待线程结束，这里做了循环即等待所有进程run结束
65          print(threads[i].getResult())
66
67      for i in nfuncs:#从这里看到，尽管run已经结束了，但进程对象还在，所以还可以调用进程对象的函数
68          threads[i].taskexec()
69          threads[i].printres()
70
71      print('all␣DONE')
72
73      print(threading.activeCount())#当前活动的线程对象的数量
74      print(threading.currentThread())#返回当前线程对象
75      print(threading.enumerate())#返回当前活动线程的列表
76
77      print('all␣DONE␣at:', ctime())
78
79  if ___name___ == '__main__':
80      main()
```

在这个脚本中，定义了三个需要计算的函数，三个函数在多线程并发时分别在各线程内计算。多线程的实现方式是用 threading.thread 线程子类的方式。三个函数的计算在构造函数内就实现了，当然也可以放到线程子类 MyThread 的 run 函数中。多线程的实现方式总共是三种:(1) 创建一个 Thread 的实例，传给它一个函数;(2) 创建一个 Thread 的实例，传给它一个可调用的类对象;(3) 从 Thread 派生出一个子类，创建一个这个子类的实例。这里采用的是第三种。

python 的线程模块包括比较底层的 thread 模块，比较高级的 threading 模块，这里使用的是后者。后者包括如下对象:

- Thread 表示一个线程的执行的对象
- Lock 锁原语对象（跟 thread 模块里的锁对象相同）
- RLock 可重入锁对象。使单线程可以再次获得已经获得了的锁（递归锁定）。
- Condition 条件变量对象能让一个线程停下来，等待其它线程满足了某个“条件”。如:状态的改变或值的改变。
- Event 通用的条件变量。多个线程可以等待某个事件的发生，在事件发生后，所有的线

程都会被激活。

- Semaphore 为等待锁的线程提供一个类似“等候室”的结构
- BoundedSemaphore 与 Semaphore 类似，只是它不允许超过初始值
- Timer 与 Thread 相似，只是，它要等待一段时间后才开始运行。

threading 模块支持守护线程，它们是这样工作的：守护线程一般是一个等待客户请求的服务器，如果没有客户提出请求，它就在那等着。如果你设定一个线程为守护线程，就表示你在说这个线程是不重要的，在进程退出的时候，不用等待这个线程退出。

如果主线程要退出的时候，不用等待某子线程完成，那就设置该线程的 daemon 属性。即，在线程开始（调用 thread.start()）之前，调用 setDaemon() 函数设定该线程的 daemon 标志（thread.setDaemon(True)）就表示这个线程“不重要”，即设置该线程为守护进程。

如果想要等待子线程完成再退出,那就什么都不用做,或者显式地调用 thread.setDaemon(False) 以保证其 daemon 标志为 False。可以调用 thread.isDaemon() 函数来判断其 daemon 标志的值。新的子线程会继承其父线程的 daemon 标志。整个 Python 会在所有的非守护线程退出后才会结束，即进程中没有非守护线程存在的时候才结束。

其中 Thread 对象的函数，包括:

- start() 开始线程的执行 (Start the thread’s activity.It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a separate thread of control.This method will raise a RuntimeError if called more than once on the same thread object.)
- run() 定义线程的功能的函数（一般会被子类重写）(Method representing the thread’s activity.You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.)
- join(timeout=None) 程序挂起，直到线程结束；如果给了 timeout，则最多阻塞 timeout 秒
- getName() 返回线程的名字
- setName(name) 设置线程的名字
- isAlive() 布尔标志，表示这个线程是否还在运行中
- isDaemon() 返回线程的 daemon 标志
- setDaemon(daemonic) 把线程的 daemon 标志设为 daemonic（一定要在调用 start() 函数前调用）

通过上述说明我们理解了 threading 模块的基本线程应用。下面我们考虑网络连接的问题

7.2 socket 模块和网络连接

网络连接需要使用 socket 模块，利用构建的 socket(即套接字) 对象进行网络连接操作。

其语法如下：socket(socket_family, socket_type, protocol=0)

socket_family 可以是 AF_UNIX 或 AF_INET。socket_type 可以是 SOCK_STREAM 或 SOCK_DGRAM。protocol 一般不填，默认值为 0。

创建一个 TCP/IP 的套接字，调用 socket.socket() 如下：

tcpSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

创建一个 UDP/IP 的套接字，调用 socket.socket() 如下：

udpSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

建立套接字对象后，就可以利用该对象的方法实现，tcp 和 udp 操作。

套接字对象的常用函数包括:

- 服务器端套接字函数
  - s.bind() 绑定地址（主机，端口号对）到套接字
  - s.listen() 开始 TCP 监听
  - s.accept() 被动接受 TCP 客户的连接，（阻塞式）等待连接的到来
- 客户端套接字函数
  - s.connect() 主动初始化 TCP 服务器连接
  - s.connect_ex() connect() 函数的扩展版本，出错时返回出错码，而不是抛异常
- 公共用途的套接字函数
  - s.recv() 接收 TCP 数据
  - s.send() 发送 TCP 数据
  - s.sendall() 完整发送 TCP 数据
  - s.recvfrom() 接收 UDP 数据
  - s.sendto() 发送 UDP 数据
  - s.getpeername() 连接到当前套接字的远端的地址
  - s.getsockname() 当前套接字的地址
  - s.getsockopt() 返回指定套接字的参数
  - s.setsockopt() 设置指定套接字的参数
  - s.close() 关闭套接字
  - s.setblocking() 设置套接字的阻塞与非阻塞模式
  - s.settimeout()a 设置阻塞套接字操作的超时时间
  - s.gettimeout()a 得到阻塞套接字操作的超时时间
- 面向文件的套接字的函数
  - s.fileno() 套接字的文件描述符
  - s.makefile() 创建一个与该套接字关连的文件

下面给出简单的 tcp 和 udp 连接示例:

7.2.1 TCP 连接

服务端:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

import sys
import socket

counter=0 #用于记录建立连接次数
socketa=socket.socket(socket.AF_INET,socket.SOCK_STREAM) #创建tcp套接字
try:
    socketa.bind(("127.0.0.1",5000))#绑定ip和端口
except socket.error:
    sys.exit("call to bind failed")
else:
    print("tcp server is ready for connection ...")

while 1:
    socketa.listen(1)#tcp套接字对象监听，等待连接
    connection,address=socketa.accept()#tcp套接字对象获得连接，Accept a connection.
    counter+=1
    print("connection",counter,"recv from:",address)
    connection.send(b'server>>>connection successful')#向连接用户发送连接成功信息
    clientmessage=connection.recv(1024)#从连接用户接收信息

    while clientmessage.decode() != "client>>>terminate":#当接收信息不是client>>>terminate时
        if not clientmessage:
            break

        print(clientmessage.decode())
        servermessage=input("server>>>")#服务端输入信息
        connection.send(("server>>>"+servermessage).encode())#并向客户端发送该信息
        clientmessage=connection.recv(1024)#发送完毕接收信息
    print("connection terminated")
    connection.close()

socketa.close()#最后关闭套接字，没有实质意义
```

客户端:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

import sys
import socket

```

```
socketb=socket.socket(socket.AF_INET,socket.SOCK_STREAM)#建立一个套接字对象
try:
    socketb.connect(("127.0.0.1",5000))#连接指定ip和端口的套接字
except socket.error:
    sys.exit(b'call to connection failed')

servermessage=socketb.recv(1024)#连接成功后，从服务端接收信息
while servermessage.decode() !="server>>>terminate":
    if not servermessage:
        break

    print(servermessage.decode())
    clientmessage=input("client>>>")#输入信息
    socketb.send(("client>>>"+clientmessage).encode())#将输入信息发送到服务端
    servermessage=socketb.recv(1024)#从服务端接收信息
print ("connection terminated")
socketb.close()
```

7.2.2 UDP 连接

服务端:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

import sys
import socket

counter=0
socketa=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)#建立用于udp协议的套接字对象
try:
    socketa.bind(("127.0.0.1",5000))#绑定ip和端口
except socket.error:
    sys.exit("call to bind failed")
else:
    print("UDP server is ready ...")

while 1:
    packet,address=socketa.recvfrom(1024)#不断从绑定的端口接收包
    counter+=1
    print("packet",counter,"recv from:",address)
    print("data from client is:",packet.decode())
    socketa.sendto(packet,address)#向客户端发送包
    print("echo sent to client...")
    if packet.decode()=="exit":#当客户端输入发送exit时，退出
        print("user want the server to exit!")
        break

socketa.close()
```

客户端:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

import sys
import socket

socketb=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)#建立用于udp协议的套接字对象

while 1:
    packet=input("packet>>>")
    print("sending packet containing",packet)
    socketb.sendto(packet.encode(),("127.0.0.1",5000))#向指定ip和端口的服务端发送包
    print("packet sent")
    packet,address=socketb.recvfrom(1024)#从指定ip和端口接收包
    print("echo recd",packet.decode())
    if packet.decode()=="exit":#退出条件
        print("user want to exit!")
        break

socketb.close()
```

注意到: 连接传输的信息是字节对象信息。

7.2.3 socketserver 的 tcp 连接

从上述两个例子可以看到，udp 协议设定监听 ip 和端口的一端是服务器，它不许要建立连接就可以同时接受来自多个用户的信息。而 tcp 协议要传输信息需要建立连接，因此需要同时与多少个用户传输信息就需要建立多少个连接，并且如果需要一直传输，这些连接还不能断开。那么只剩下一种方式可以同时建立多个连接，即用多线程的方式，一个线程建立一个连接。

在 python 标准库中有一个 socketserver 模块，利用其封装的高级 socket 对象，可以轻松的建立 tcp 服务端，也包括多线程的 tcp 服务端。下面看一些例子，首先是单线程的 tcp 服务端:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

from socketserver import (TCPServer as TCP,StreamRequestHandler as SRH)

from time import ctime

HOST = "127.0.0.1"
PORT = 9999
ADDR = (HOST, PORT)

```

```
class MyRequestHandler(SRH):
    def handle(self):
        while 1:
            self.data = self.rfile.readline() #.strip()有时候不确定传递来的数据里面是否有回车，那么可以用strip 函数把前后空格去掉，然后在字符串中增加“\n” 来实现换行。
            print("{} wrote:".format(self.client_address[0]))
            print(self.data.decode())
            self.wfile.write(self.data) #.upper()

tcpServ = TCP(ADDR, MyRequestHandler)
print('waiting for connection...')
tcpServ.serve_forever()
```

原理很简单，构建一个 TCPServer 对象，并定义一个 StreamRequestHandler 子类用于响应请求。注意响应函数 handle 中的接收信息和发送信息函数分别是 rfile.readline 和 wfile.write。用户端与前面的例子类似，注意其中直接对象的两种表示方式，一种是用 encode 得到，一种是利用 bytes 得到。

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

import sys
import socket

socketb=socket.socket(socket.AF_INET,socket.SOCK_STREAM)#建立一个套接字对象
try:
    socketb.connect(("127.0.0.1",9999))#连接指定ip和端口的套接字
except socket.error:
    sys.exit(b'call to connection failed')
else:
    print("connected")

#socketb.sendall(bytes("hello" + "\n", "utf-8"))
#received = str(socketb.recv(1024), "utf-8")
socketb.sendall("client>>>is connected \n".encode())#将输入信息发送到服务端
servermessage=socketb.recv(1024)#连接成功后，从服务端接收信息
while servermessage.decode() !="server>>>terminate":
    if not servermessage:
        break

    print(servermessage.decode())
    clientmessage=input("client>>>")#输入信息
    #socketb.sendall(bytes(clientmessage+"\n", "utf-8")) #将输入信息发送到服务端
    socketb.send(("client>>>"+clientmessage+"\n").encode())#正确的原因是在于多了一个\n，send和sendall都可以。
    servermessage=socketb.recv(1024)#从服务端接收信息
print ("connection terminated")
socketb.close()
```

下面看一个多线程的例子，服务端与单线程的类似:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_

import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

class MyTCPHandlera(socketserver.StreamRequestHandler):

    def handle(self):
        while 1:
            # self.rfile is a file-like object created by the handler;
            # we can now use e.g. readline() instead of raw recv() calls
            self.data = self.rfile.readline().strip()
            print("{} wrote:".format(self.client_address[0]))
            print(self.data)
            # Likewise, self.wfile is a file-like object used to write back
            # to the client
            self.wfile.write(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    # Create the server, binding to localhost on port 9999
    #server=socketserver.TCPServer((HOST, PORT), MyTCPHandlera)
    server=socketserver.ThreadingTCPServer((HOST, PORT), MyTCPHandlera)
    server.serve_forever()
```

看到，多线程和单线程的差异仅在构建 tcpserver 对象上。还注意到，响应类可以是 StreamRequestHandler，也可以是 BaseRequestHandler 类。对于用户端，下面的代码与前面给出的例子是类似的。

```
import socket
import sys

HOST, PORT = "localhost", 9999
#data = " ".join(sys.argv[1:])
data="hello server!"

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
```

```
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

    print("Sent:     {}".format(data))
    print("Received: {}".format(received))

    servermessage=data
    while servermessage !="server>>>terminate":
        if not servermessage:
            break

        print(servermessage)
        clientmessage=input("client>>>")#输入信息
        sock.sendall(bytes(clientmessage+"\n", "utf-8"))
        servermessage = str(sock.recv(1024), "utf-8")
        #sock.sendall(("client>>>"+clientmessage).encode())#将输入信息发送到服务端
        #servermessage=sock.recv(1024)#从服务端接收信息
```

7.2.4 自定义的多线程 tcp 服务

上述用 tcpserver 模块的方法实现了多线程的 tcp 服务，但对于理解多线程的运作并没有多大用处。因此我们可以自己来构建多线程的 tcp 服务:

```
#!/usr/bin/env python3
#_*_coding: utf-8 _*_


import threading
import random
from time import sleep, ctime
import sys
import socket


class MyThread(threading.Thread):
    def ___init___(self,connection,address):
        threading.Thread.___init___(self)
        self.connection = connection
        self.address = address
        print("connection created","from:",self.address)
        self.connection.send(b'server>>>connection successful')
    def run(self):
        while 1:
            clientmessage=self.connection.recv(1024)
            print(clientmessage.decode())
            if (not clientmessage) or (clientmessage.decode() == "client>>>terminate"):
                print("connectiong is closed")
```

```
25                  self.connection.send("server>>>terminate".encode())
26                  self.connection.close()
27                  break
28              self.connection.send(("server>>>echoed").encode())
29
30
31  socketa=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
32  try:
33      socketa.bind(("127.0.0.1",5000))
34  except socket.error:
35      sys.exit("call to bind failed")
36  else:
37      print("socket created,waiting for connection ...")
38
39  threads = []
40  while 1:
41      socketa.listen(2)
42      connection,address=socketa.accept()
43      t = MyThread(connection,address)
44      threads.append(t)
45      t.start()
46
47  socketa.close()
48
49  def main():
50      pass
51
52
53  print(threading.activeCount())#当前活动的线程对象的数量
54  print(threading.currentThread())#返回当前线程对象
55  print(threading.enumerate())#返回当前活动线程的列表
56
57
58  if __name__ == '__main__':
59      main()
```

对于这个服务端，上述给出的所有的 tcp 用户端都可以连接。

Chapter 8

理解字符编码及在文本编辑器、latex、python 中的表现

之所以想要把字符编码问题搞清楚，是因为在学习 python 过程中遇到了一个问题。在 dive in python(深入 python 2.x 版) 的第 5 章中，有一个 fileinfo 源代码，为理解它的代码并进行测试，我首先将代码转换到 python3.x 版 (因为我安装的是 3.6.2 版)。主要的修改包括: print 语句，MP3 文件路径，以及 Userdict 基类换成 dict，(其中有一个难点是: 修改后代码中 fileinfo 类继承的 dict 类没有 setitem 这一方法，不知道要怎么改。要解决它需要理解类的继承，特别是子类中重新实现的方法中调用一下父类的原方法的原因，是因为父类的一些参数也需要调用该方法进行设置)。修改完成后程序能输出结果，但是输出的只有文件名这一信息，这与希望给出的结果不符，于是开始需找其中的原因。

从代码可以看到，在 MP3 文件的末尾通常存在以 TAG 开头的一段 128 字节的信息用于存储 mp3 的如专辑等其他信息，fileinfo 实现的功能就是解析这段信息，然后输出这一段信息，可以知道的是这一段信息是 byte(字节) 类信息，这一段信息的解析和显示需要对字节的信息进行解码，那么之所以出现问题很可能来自该信息中文字符编码问题。

首先我们了解一下汉字 (或 cjk 字符) 的 unicode 范围，这可以通过查 unicode 表知道，见表8.1，在 latex 中 cjk 字符的判断通常从 2E80 开始。

下面来分析一下前述字节信息可能存在的编码问题:

8.1 MP3 文件 TAG 信息的编码

对于 MP3 文件，在 windows 操作系统中，如果利用属性/详细信息设置把 MP3 的艺术家、唱片、专辑等信息改成为拼音 (即用英文字符) 来表示，那么 print 输出显示是正常的拼音。而如果改回中文即输出字节内的字符存储信息。比如，《约定》这首歌，改成拼音后其中某段信息是 yueding，若是改回中文则是\xd4\xbc\xb6\xa8，这是一段字节信息 (bytes 对象)。因

表 8.1: 汉字的 unicode 范围

编码范围	字符类型
2E00-2E7F	追加标点
2E80-2EFF	cjk 部首补充
2FF0-2FFF	表意文字描述符
3000-303F	cjk 符号和标点
3300-33FF	cjk 兼容
3400-4DBF	cjk 统一表意符号扩展
4E00-9FBF	cjk 统一表意符号
20000-2F8BF	中日韩统一表意文字扩展 B

为输入的中文“约定”是知道的，因此很容易判断这一段字节内的存储信息是“约定”的字符编码。下面我们就要看看这到底是什么编码？

8.2 latex 中的字符编码

因为平时用 latex 比较多，很容易想到用 latex 来查看/显示字符或者其对应的编码。于是首先利用 latex 做一个测试。

在 latex 中十进制的字符编码用整数表示，十六进制的编码用双引号加编码表示，因为 latex 文档采用 UTF-8 字符格式存储，所以 tex 中利用`字符所显示的是字符编码是 unicode 码。

(这里面有个概念需要区分，字符存在多种编码方式比如 *unicode*，*gbk* 等，而文档的字符存储也有多种格式，比如 *utf-8* 格式，该格式存储的字符的实际信息是字符的 *unicode* 编码，又比如采用 *gb2312* 格式存储的文件的实际信息是字符的 *gb2312* 编码。有的字符编码有多种存储格式，比如 *unicode* 编码可以用 *utf-8* 存储，也可以用 *utf-16* 存储。换一种说法就是 *utf-8* 格式 *(*或编码*)* 其实是 *unicode* 码的一种实现，即用 *utf-8* 码表示 *unicode* 码。我个人的理解，*utf-8* 从本质上来说是一种字符在字节中存储的二进制码的编码方式，这其实可以称为是一种存储格式，当然说成编码也没有问题。在我后面的讨论中格式和编码是相同的概念。)

测试如下:

十进制整数编码 98 对应的字符为:b

十六进制整数编码 26 对应的字符为:&

十进制整数编码 20013 对应的字符为: 中

字符 b 对应的十进制整数编码为: 98

字符 & 对应的十进制整数编码为: 38

字符中对应的十进制整数编码为: 20013

字符约对应的十进制整数编码为: 32422
字符定对应的十进制整数编码为: 23450
十六进制整数编码 D4BC 对应的字符是:
十六进制整数编码 B6A8 对应的字符是:
十六进制整数编码 7EA6 对应的字符是: 约
十六进制整数编码 5B9A 对应的字符是: 定
十六进制整数编码 4E2D 对应的字符是: 中
十六进制整数编码 6587 对应的字符是: 文

注意到，字节信息\xd4\xbc\xb6\xa8实际对应的是约定这两个字符，那么这两个字符对应的编码是应该为 D4BC 和 B6A8，但从 latex 输出看，显然这两个编码对应的字符不是“约定”，说明这两个编码不是 unicode 码，从输出看约定的 unicode 编码是 7EA6 和 5B9A。所以很容易联想到这可能是操作系统的默认编码，而 windows 一般是 gb2312 编码。我们可以利用文本编辑工具来看一下是不是:

8.3 文本编辑器查看字符编码

利用 notepad++ 或者 notepad2 文本编辑器，很容易生成不同存储格式的 3 个文件，内容均为约定两个字符，第一个文件 a 采用默认存储格式，第二个文件 b 采用 gb2312 格式，第三个文件 c 采用 utf-8 格式。

利用 ultraedit 编辑器打开三个文件，并且利用十六进制编辑器查看文档存储的字节信息，可以看到:

```
文件a:
00000000h: D4 BC B6 A8 ; 约定

文件b:
00000000h: D4 BC B6 A8 ; 约定

文件c:
00000000h: E7 BA A6 E5 AE 9A ; 绾﹀畾
```

其中文件 ab 内容相同，说明系统默认存储格式为 gb2312，而文件 c 内容看到两个字符采用了 6 个字节表示，一个字符分别是 3 个字节，这就是 utf-8 的存储格式，而因为系统默认显示字符存储格式也是 gb2312，因此系统会根据两个字节的信息来显示字符于是就得到“绾﹀畾”这三个字符。可以想见，如果系统默认的字符存储格式是 utf-8，那么文件 c 显示的字符将是约定。退出十六进制编辑器，查看文件 c 中两个字符的字符属性可以看到:

```
对于约
十进制值32422
十六进制值0x7ea6

```

```
对于定
十进制值23450
十六进制值0x5b9a
```

到这里可以清楚知道，系统默认采用 gb2312 格式对汉字进行存储。因此在 python 中如果要正确处理 MP3 的 tag 信息，那么就需要进行特殊的处理，即字节解码的时候要用相应的编码格式如 gb2312，而不仅仅采用默认的 utf-8 格式。

8.4 python 中的字符编码

对于 python3，默认的字符串编码是 unicode 编码，默认的存储格式 utf8，所以处理 gb2312 编码的字符等特殊的情况就需要特殊处理。

在 python 中，表示字符串的是字符串对象，表示存储信息的是字节对象。一般处理围绕这两类对象进行。我们首先看一下 python 中的一些编码相关的工具:

查看字符对象对应的字节存储对象，利用字符对象的 encode 方法，比如:

```
>>> "约定".encode("gbk")
b'\xd4\xbc\xb6\xa8'
>>> "约定".encode("gb2312")
b'\xd4\xbc\xb6\xa8'
>>> "约定".encode("utf-8")
b'\xe7\xba\xa6\xe5\xae\x9a'
```

查看字节存储对象对应的字符串对象，利用字节对象的 decode 方法，比如:

```
>>> b'\xe7\xba\xa6\xe5\xae\x9a'.decode("utf-8")
'约定'
>>> b'\xd4\xbc\xb6\xa8'.decode("gb2312")
'约定'
```

通过这两个方法可以在字符串对象和存储的字节对象之间互相转化。

查看 unicode 字符的 unicode 编码用函数 ord()，hex() 将整数转化为 16 进制数字符串，oct() 将整数转化为 8 进制数字符串，bin() 将整数转化为 2 进制数字符串，int() 将字符串参数转化为整数，str() 将参数转换为字符串。比如:

```
a=ord("中")
>>> a
20013
>>> hex(a)
'0x4e2d'
>>> oct(a)
'0o47055'
>>> bin(a)
'0b100111000101101'
>>> int(0x4e2d)
20013
```

```
>>> int(0o47055)
20013
>>> int(0b100111000101101)
20013
>>>
```

通过这些函数可以得到字符的 unicode 编码，也可以在各种不同进制的编码之间做转换。

如前所述，python3 中字符与编码的转换主要依靠字符串和字节对象来进行处理，在基本的应用中使用 encode 和 decode 已经足够。事实上字符的编码问题在 python 中是比较常见的问题，colcloud 的python3 字符编码和微寒的python3 字符编码问题两篇博客都讲的是字符编码导致的相关问题。

关于 python3 的字符编码问题，的python3 如何解决字符编码问题详解和 andrewleeeeee 的python3 字符编码这两篇博客也给出了比较详细的说明。

其中需要注意的是 getdefaultencoding 命令得到的默认编码是 python 的字节 (存储) 编码，而不是 windows 操作系统的，比如:

```
>>> import sys
>>> print('default encoding is:',sys.getdefaultencoding())
default encoding is: utf-8
```

其中还有两个 python 函数 type() 和 len()，在字符编码问题中可能会比较有用，type() 用于查看对象的类型，len() 输出对象的长度，比如:

```
>>> code='中'.encode()
>>> type(code)
<class 'bytes'>
>>> len(code)
3
>>> code
b'\xe4\xb8\xad'
>>> char=b'\xe7\xba\xa6'.decode()
>>> type(char)
<class 'str'>
>>> len(char)
1
>>> char
'约'
>>>
```

为进一步了解，下面我们来看一看字符与 unicode 编码以及 utf-8 格式的关系:

8.5 python 中的字符与 unicode 码、utf-8 字节码的相互转换

首先介绍一下 utf-8 编码的规则:

utf-8 格式规则主要包括两条 (参考: 阮一峰的字符编码笔记:ASCII,Unicode 和 UTF-8):

(1) 对于单字节的符号，字节的第一位设为 0，后面 7 位为这个符号的 unicode 码

(2) 对于 n 字节符号 (n>1)，第一个字节的前 n 位都设为 1，第 n+1 位设为 0，后面字节的前两位一律设为 10。剩下的二进制位，从后向前填入字符 unicode 码的二进制数，多出的用 0 补充。

需要注意: 在 python3 中，单字节的字符 utf-8 编码的二进制数通常是少于 8 位的，即高位的 0 是不给出的，在处理时要小心。

在前一节我们介绍过使用 ord() 函数可以得到字符的 unicode 码，那么能否从 unicode 码得到字符呢？这其实也是很容易得到的。利用字符串字面常量 (参见 *python3.6.2.chm 文档 2.4 Literals*) 即可实现，比如:

```
>>> ord("国")
22269
>>> hex(ord("国"))
'0x56fd'
>>> '\u56fd'
'国'
>>>
```

该例表明 python 可以很容易的利用 unicode 码表示字符，或者得到字符的 unicode 码。这比查Unicode 编码表要来的方便。但这种直接使用字符字面常量获得字符的方法在程序使用中会有一些局限，因为很多时候无法直接输入一个字面常量，那么是否可以采用函数方法根据输入参数来返回对应的转换呢，包括从字符到 unicode 码，从字符到 utf-8 字节码，从 unicode 码到字符，从 utf-8 字节码到字符，从 utf-8 字节码到 unicode 码，从 unicode 码都 utf-8 字节码，答案当然是肯定的。下面我们来做这个事情:

首先给出从 utf-8 字节码到 unicode 码的转换函数，采用两种方法，一种是利用 python 工具，另一种是根据上述给出的 utf-8 编码规则:

方法 1:

```
def bytestounicodeb(byteobj):
    str=byteobj.decode()#以默认编码格式解析字节信息为字符
    ucode=hex(ord(str)) #得到字符的整数编码并转化为16进制数表示的字符串
    print("char's bytes is",byteobj)
    print('char is',str)
    print("char's code is %d, base=10" % ord(str))
    print("char's code is %s, base=16" % ucode)
    print("char's code is %s, base=2" % bin(ord(str)))
```

方法 2:

```
def bytestounicode(byteobj):#验证utf-8格式
    str=byteobj.decode()#以默认编码格式解析字节信息为字符
    str0=bin(byteobj[0])#得到第一个字节的二进制数构成的字符串
    nbytes=len(byteobj)#也可以用下面注释这一段来判断
    # nbytes=0
    # if len(str0)<10: #单字节字符高位的0不给出，所以加上0b后字符串位数会小于10
```

```
    # strucode=str0
    # else:
    # for i in range(2,10):
    # if str0[i]==”0”:
    # break
    # else:
    # nbytes+=1
    if nbytes>1:
        strucode='0b'+str0[2+nbytes:]#二进制字符前面加’0b’或者不加对于类型转换没有影响，这里为了显示效果加上它
        for i in range(1,nbytes):
            strucode+=bin(byteobj[i])[4:]#不是第一个字节则取0b10后的字符，即从第5个字符(第4个索引)开始
    else:
        strucode=str0
    print("char's bytes is",byteobj)
    print('char is',str)
    print("char's code is %d, base=10" % int(strucode,base=2))
    print("char's code is %s, base=16" % hex(int(strucode,base=2)))
    print("char's code is %s, base=2" % strucode)
```

接着给出从 unicode 码都 utf-8 字节码或字符的函数，也给出两种，一种完全利用字符串操作，一种则利用整数的位操作

方法 1:

```
def unicodetochar(numobj):#利用二进制字符串的操作来得到utf8格式编码
    if numobj < 0x007f:#确定字节数
        nbytes=1
    elif numobj < 0x07ff:
        nbytes=2
    elif numobj < 0xffff:
        nbytes=3
    elif numobj < 0x10ffff:
        nbytes=4
    else:
        print("error")
    ucodestr=bin(numobj)
    blist=[]
    for i in range(nbytes):#处理得到各个字节的信息
        start=len(ucodestr)-6-6*i #不处理最前面一个字节时，取6个字符
        end=len(ucodestr)-i*6
        if i==nbytes-1: start=2 #处理最前面一个字节时，取去掉0b剩下的字符
        #print(start,end,ucodestr[start:end])
        if i==nbytes-1:
            strheader='11111111'[:nbytes]
            strheader+='00000000'[:8-nbytes-(end-start)]
            # for j in range(nbytes): #上述两句用for循环也可以
            # strheader+=’1’
            # numzero=8-nbytes-(end-start)
            # for j in range(numzero):
            # strheader+=’0’
            blist.insert(0,strheader+ucodestr[start:end])#存入blist列表中
```

```
28          else:
29              blist.insert(0,'10'+ucodestr[start:end])
30      strhex="".join(hex(int(elem,base=2))[2:] for elem in blist)#这里利用了join方法连接字符串
31      print("char's code is %d, base=10" % numobj)
32      print("char's code is %s, base=16" % hex(numobj))
33      print("char's code is %s, base=2" % ucodestr)
34      print("char's byte string is",strhex)
35      print("char's bytes is",bytes.fromhex(strhex))#fromhex的参数只要是由2个16 进制的数的字符串构成即可
36      print("char is",bytes.fromhex(strhex).decode())
```

其中，首先根据 unicode 值的大小来确定 utf-8 编码的字节数，然后在对 unicode 值的二进制数字符串进行处理。utf-8 的字节数划分范围见表8.2(参见 *python3.6.2.chm* 文档 *7.2.2. Encodings and Unicode*):

表 8.2: utf-8 编码字节数划分范围

Range	Encoding
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

方法 2:

```
1  def unicodetocharb(numobj):#利用整数的位的操作来得到utf8格式编码
2      if numobj < 0x007f:#确定字节数
3          nbytes=1
4          ref=0 #00000000用于1字节的位或操作
5      elif numobj < 0x07ff:
6          nbytes=2
7          ref=0xc080 #1100000010000000用于2字节的位或操作
8      elif numobj < 0xffff:
9          nbytes=3
10         ref=0xe08080 #111000001000000010000000用于3字节的位或操作
11     elif numobj < 0x10ffff:
12         nbytes=4 #11110000100000001000000010000000用于4字节的位或操作
13         ref=0xf0808080
14     else:
15         print("error")
16     res=ref
17     src=numobj
18     for i in range(nbytes):#遍历nbytes个字节，顺序是从后往前
19         if i==nbytes-1:#处理最前面一个字节时剩下的位全部取出用于位或
20             a=src<<(i*8)
21             res=res |a
22         else:
23             a=src & 0x3f #不处理最前面一个字节时，取6位用于位或
24             src=src>>6 #源整数中，6位取出后直接丢弃
```

```
25              a=a<<(8*i) #把取出的6为放到i字节上用于位或
26              res=res | a
27      #print ("%x" %res)
28      strhex=("%x" %res)#这里利用了printf方式的字符串转换
29      print("char's code is %d, base=10" % numobj)
30      print("char's code is %s, base=16" % hex(numobj))
31      print("char's code is %s, base=2" % bin(numobj))
32      print("char's byte string is",strhex)
33      print("char's bytes is",bytes.fromhex(strhex))
34      print("char is",bytes.fromhex("%x" %res).decode())
```

第二种方法中采用位操作的思路，参考了 zqiang3 给出的unicode 编码转 utf-8 编码。

8.6 小结

利用 python 提供的工具基本能够满足字符与其编码的转换，如果在一些应用中需要函数来返回一些参数，可以利用文中给出的 unicode 到字符的转换函数。

ps: 详细测试程序见unicodetest.py