Python 3 标准库教程

目录

1 Python 3 输入和输出

1.1 `input()`输入

Python 3 仅保留了`input()`函数，它可接收任意任性输入，将所有输入默认为字符串处理，并返回字符串类型。

执行下面的程序在按回车键后就会等待用户输入：

```
#!/usr/bin/env python3
input("\n\n按下 enter 键后退出。")
```

上述代码中 ，`'\n\n'`在结果输出前会输出两个新的空行。一旦用户按下 `Enter` 键时，程序将退出。

1.2 `print()`输出

Python 中有两种**输出值**的方式: **表达式语句**和`print()`**函数**。第三种方式是使用**文件对象**的`write()`**方法**，标准输出文件可以用`sys.stdout`引用。

- 若希望输出的形式更加多样，可以使用`str.format()`函数来格式化输出值 (见**Section 4.5.3**)。
- 若希望将输出的值转成字符串，可以使用`repr()`或`str()`函数来实现。
    - `str()`: 函数返回一个用户易读的表达形式。
    - `repr()`: 产生一个解释器易读的表达形式。

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

两种方式输出一个平方与立方的表：

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))  # 使用rjust()方法将字符串靠右,并在左边填充空格
...
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
```

1.3 读和写文件

1.3.1 打开文件`open()`函数

利用`open()`函数将会返回一个`file`对象，基本语法格式如下:

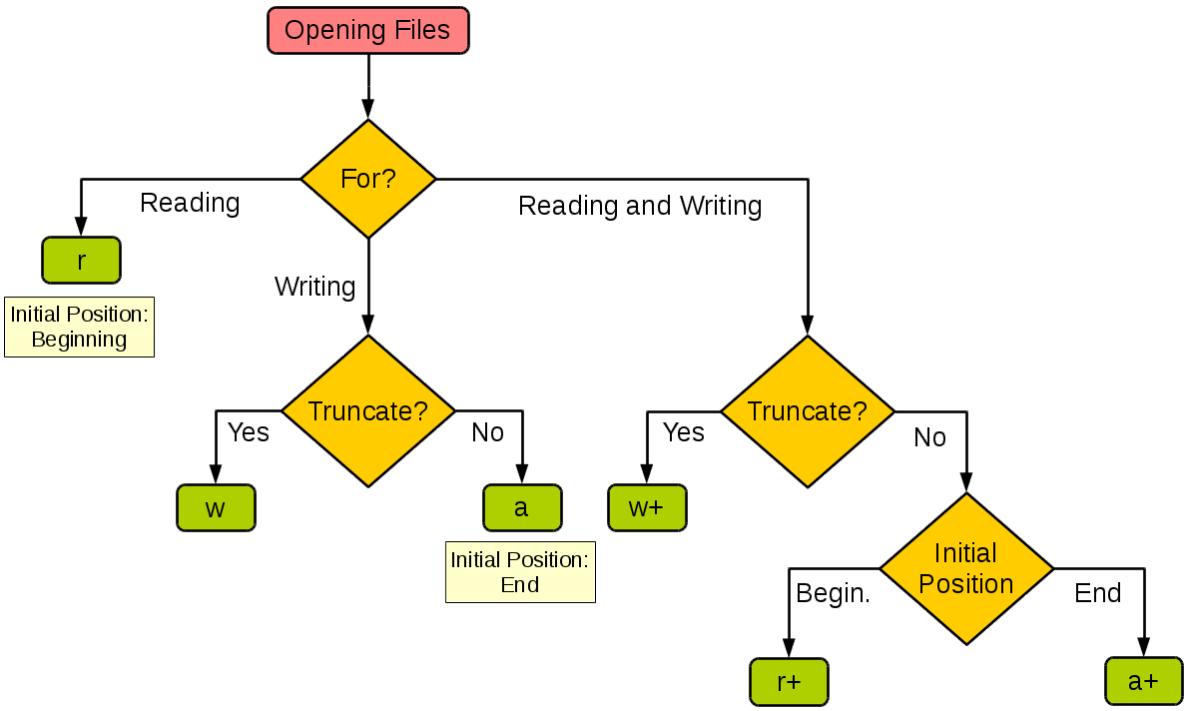
```
open(filename, mode='r', encoding=None)
```

- `filename`：包含了你要访问的文件名称的字符串值。
- `mode`：决定了打开文件的模式：只读，写入，追加等。该参数是非强制的，默认文件访问模式为只读。
- `encoding`：一般使用`utf-8`

不同模式打开文件的完全列表：

Mode	Description
t	(默认) 文本模式
b	二进制模式
+	打开一个文件进行更新(可读可写)。
r	(默认) 以只读方式打开文件。文件的指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，则创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，则创建新文件进行写入。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，则创建新文件。
w+	打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，则创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，则创建新文件。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，则创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，则创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，则创建新文件用于读写。

下图很好的总结了这几种模式：



Mode	r	r+	w	w+	a	a+
读	+	+		+		+
写		+	+	+	+	+
创建			+	+	+	+
覆盖			+	+		
指针在开始	+	+	+	+		
指针在结尾					+	+

1.3.2 文件对象的方法

本节中剩下的例子假设已经创建了一个称为`f`的文件对象。

1. `f.read([size])`函数：读取一定数目的数据，然后作为字符串或字节对象返回。`size`是一个可选的数字类型的参数，当`size`被忽略了或者为负时该文件的所有内容都将被读取并且返回。

```
#!/usr/bin/env python3
f = open("/tmp/foo.txt", "r")  # 打开一个文件
str = f.read()  # 读取整个文件
print(str) # 输出读取内容
f.close()  # 关闭打开的文件
```

2. `f.readline([size])`函数：从文件中读取单独的一行，换行符为`'\n'`。若`f.readline()`返回一个空字符串, 则说明已经读取到最后一行。
3. `f.readlines([size])`函数：将返回该文件中包含的所有行(直到结束符`EOF`)，若碰到结束符`EOF`则返回空字符串。若设置可选参数`size`，则读取指定长度的字节, 并且将这些字节按行分割。

   若文件"runoob.txt"的内容如下：

```
1:www.runoob.com
2:www.runoob.com
3:www.runoob.com
4:www.runoob.com
5:www.runoob.com
```

循环读取文件的内容：

```
#!/usr/bin/env python3
fo = open("runoob.txt", "r")  # 打开文件
print("文件名为: ", fo.name)  # 输出文件名

for line in fo.readlines():             # 依次读取每行
    line = line.strip()                 # 去掉每行头尾空白
    print ("读取的数据为: %s" % (line))  # 输出每行内容

fo.close()  # 关闭文件
```

4. 另一种方式是**迭代一个文件对象**然后**读取每行**：

```
#!/usr/bin/env python3
f = open("/tmp/foo.txt", "r")  # 打开一个文件
for line in f:  # 迭代一个文件对象然后读取每行
    print(line, end='')
f.close()  # 关闭打开的文件
```

5. `f.write(string)` 函数：将 `string` 写入文件中，返回写入的字符数。若想写入非字符串，则需先进行转换。
6. `f.writelines(list)` 函数：向文件写入一个序列字符串列表，若要换行则需自行加入每行的换行符。

```
#!/usr/bin/env python3
fo = open("test.txt", "w")  # 打开文件
print("Filename is: ", fo.name)  # 输出文件名
seq = ["菜鸟教程 1\n", "菜鸟教程 2"]  # 字符串列表（含换行符）
fo.writelines( seq )  # 写入文件
fo.close()  # 关闭文件
```

7. `f.tell()` 函数：返回文件对象当前所处的位置，即从文件开头开始算起的字节数。
8. `f.seek(offset, from_what)` 函数：可以使用该函数改变文件当前的位置。

   `from_what` 的值，若是 0 表示开头 (默认)；1 表示当前位置； 2 表示文件的结尾，例如：

   - `seek(x,0)`： 从文件首行首字符开始往后移动 `x` 个字符
   - `seek(x,1)`： 表示从当前位置往后移动 `x` 个字符
   - `seek(-x,2)`：表示从文件的结尾往前移动 `x` 个字符

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

9. **`f.truncate(size)`**函数：用于从文件的首行首字节开始截断，截断文件为`size`个字节，无`size`则表示从文件开头开始截断到当前位置；截断之后后面的所有字节被删除。实例：

```
#!/usr/bin/env python3
fo = open("runoob.txt", "r+", encoding="utf-8")
print ("文件名: ", fo.name)

fo.seek(36)
fo.truncate()  # 从第36个字节以后的内容全部删除了

fo.seek(0,0)
line = fo.readlines()
print("读取行: %s" % (line))

fo.truncate(10)  # 截取10个字节
fo.seek(0,0)
str = fo.read()
print("读取数据: %s" % (str))

fo.close() # 关闭文件
```

以上实例输出结果为：

```
文件名:  runoob.txt
读取行: ['1:www.runoob.com\n', '2:www.runoob.com\n', '3:']
读取数据: 1:www.runo
```

10. **`f.close()`**函数：在文本文件中 (那些打开文件的模式下没有`b`的)，只会相对于文件起始位置进行定位。当处理完一个文件后，调用`f.close()`来关闭文件并释放系统资源，若尝试再调用该文件则会抛出异常。
11. **`f.fileno()`**函数：返回一个整型的文件描述符 (file descriptor)，可用于底层操作系统的 I/O 操作。
12. 当处理一个文件对象时，**使用`with`关键字**是非常好的方式。在结束后，它会帮你正确的关闭文件。而且写起来也比`try - finally`语句块要简短：

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

1.3.3 使用`with...as`语句

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。下面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上:

```
file = open("myfile.txt")
for line in file.readlines():
    print(line, end="")
file.close()
```

以上这段代码的问题是，如果读取文件过程中出现错误，那么文件会保持打开状态一直占用内存资源。要解决此类问题，可以使用`try..finally`语句来实现自动关闭文件。而关键词`with`语句则提供了更为简单的实现，可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法。

实例1：(以下这段代码执行完毕后，就算在处理过程中出了问题，文件`file`也总会被关闭)

```
with open("myfile.txt") as file:
    for line in file:
        print(line, end="")
```

实例2：(使用`with`语句来打开文件读写、用`pickle`包完成数据的存储及恢复的操作)

```
import pickle
try:
  with open("phone.pickle", "wb") as outf:
    pickle.dump("13193388105", outf)
  except:
    print("file have error.")

try:
  with open("phone.pickle", "rb") as outf:
    data = pickle.load(outf)
    print(type(data))
    print(data)
  except:
    print("file have error.")
```

1.3.3.1 `with`语句的执行原理

`with`语句的基本形式是：

```
with expression1 as var1 [,expression2 as var2 [...]]:
  code_block
```

这样的一段代码可以称为一个上下文 (context)，在执行`with`语句时，解释器会先求出表达式的值，这个值 (对象) 就是一个上下文管理器，并且假设这个对象拥有如下类的构造方法：

```
def __enter__():
  # 描述进入上下文的动作
  pass

 def __exit__():
  # 描述退出上下文的动作
  pass
```

`with`语句在求出这个上下文管理器对象之后，会自动执行`__enter__()`，并将这个对象的返回值赋值于`as`之后的变量，然后执行语句块。最终在退出上下文前，解释器会自动执行对象的`__exit__()`。Python 中的系统和标准库的一些类型定义了这对操作，可以直接用于`with`语句，比如文件对象的操作。

1.3.3.2 自定义类的上下文管理

若自己有类似的计算过程需要抽取出来，那么可以自定义一个类，并且包含`__enter__()`和`__exit__()`方法。

```
class File(object):
  def __init__(self, file_name, method):
    self.file_obj = open(file_name, method)
  def __enter__(self):
    return self.file_obj
  def __exit__(self, exception_type, exception_value, traceback):
    self.file_obj.close()
    # 异常处理的代码块 #

with File('demo.txt', 'r') as opened_file:
  for line in opened_file:
    print(line)

# with 语句执行的步骤：
# (1) 创建一个 File 类的实例，先调用__init__方法来使用指定模式打开一个指定的文件
# (2) with 语句调用 File 类的__enter__方法返回打开的文件对象的句柄
# (3) 打开的文件句柄被传递给 opened_file，再执行 with 语句内的代码块
# (4) 若执行过程中有异常或执行完毕无异常，with 语句调用 File 类的__exit__方法
# (5) File 类的__exit__方法关闭了文件

# with 语句处理异常的步骤：
# (1) 将异常的 type，value 和 traceback 传递给__exit__方法
# (2) 让__exit__方法来处理异常
# (3-1) 若__exit__返回的是True，那么这个异常被忽略；
# (3-2) 若__exit__返回的是True以外的东西，那么这个异常将被 with 语句抛出
```

此外，还可使用上下文管理包 (contextlib) 及装饰器和生成器来实现，则不需要用到类构造方法：

```
from contextlib import contextmanager  # 引入上下文管理器

@contextmanager  # 给函数引入装饰器
def myopen(filename, mode):
  file = open(filename, mode, encoding='utf-8')
  try: # 上文
    yield file
  except Exception as err:
    print('Errpr: ', err)
```

```
  finally: # 下文
    file.close()

with myopen("demo.txt", 'r') as fobj:  # 把 try 中的 yield 中的 file 赋值给 fobj
  # with 会将后面的函数中的 yield 赋值给 fobj
  for line in fobj:
    print(line)
  # 等待上面的循环结束后,才最终执行 finally 的代码，所以这就是上下文管理
```

1.3.4 使用`pickle`模块

Python 中的`pickle`模块实现了基本的数据序列化和反序列化操作。通过`pickle`模块的序列化能够将程序中运行的对象信息保存到文件中并永久存储。通过`pickle`模块的反序列化，能够从文件中创建上一次程序保存的对象。**基本接口**：

```
pickle.dump(obj, file, [,protocol=None])  # 将对象的pickled表示写入数据文件
pickle.dumps(obj [,protocol=None])        # 将对象的pickled表示作为bytes对象返回

obj      : 要封装的对象
file     : 必须以二进制可写模式即"wb"打开
protocol: 使用的协议：0,1,2,3,4,5,-1（若为负数,则选用最高）

>>> pickle.HIGHEST_PROTOCOL # 可用的最高协议的版本（Python3.8中为5）
5
>>> pickle.DEFAULT_PROTOCOL # 默认的协议版本（Python3.8中默认为4）
4
>>> pickle.format_version  # 当前格式版本
'4.0'
>>> pickle.compatible_formats  # 兼容的格式版本
['1.0', '1.1', '1.2', '1.3', '2.0', '3.0', '4.0', '5.0']
>>>
>>> with open(filename, 'wb') as file:
...   pickle.dump(data, file)
...
>>> p_str = pickle.dumps(data); print(p_str)
```

有了`pickle`这个对象，就能对`file`以读取的形式打开:

```
x = pickle.load(file)  # 从数据文件中读取转换被封装的对象并返回为Python的数据结构
x = pickle.loads(obj)  # 从字节对象中读取转换被封装的对象并返回为Python的数据结构

file: 必须以二进制可读模式即"rb"打开

>>> with open(filename, 'r') as file:
...   data = pickle.load(file)
...
>>> mes = pickle.loads(p_str); print(mes)
```

使用`pickle`模块可能出现**三种异常**：

1. `PickleError`：封装和拆封时出现的异常类，继承自`Exception`
2. `PicklingError`：遇到不可封装的对象时出现的异常，继承自`PickleError`
3. `UnPicklingError`：拆封对象过程中出现的异常，继承自`PickleError`

实例1：

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

# 原始数据对象1 (字典)
data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}
# 原始数据对象2 (递归引用)
selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

# 使用pickle模块将数据对象保存到文件
output = open('data.pkl', 'wb')

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

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

# 关闭文件
output.close()
```

实例2：

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

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

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

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

pkl_file.close()
```

2 Python 3 OS 文件/目录方法

Python 中的`os`模块的部分**常用属性**如下所示：

1. **os.curdir**: 调用`os`模块获取当前路径的符号，即" . "。
2. **os.pardir**: 调用`os`模块获取上层路径的符号，即" .. "。
3. **os.sep**: 调用`os`模块获取路径中的分隔符，即" / "。
4. **os.linesep**: 调用`os`模块获取换行符，即" /n "。

Python 中的`os`模块提供了非常丰富的方法用来处理文件和目录。**常用的方法**如下所示：

1. **os.access(path, mode)**: 用于检验权限模式。
    - **path** -- 要用来检测是否有访问权限的路径
    - **mode** -- 默认为`os.F_OK`
        - **os.F_OK:** 作为access()的mode参数，测试path是否存在。

    - **os.R_OK:** 包含在access()的mode参数中 ， 测试path是否可读。
    - **os.W_OK** 包含在access()的mode参数中 ， 测试path是否可写。
    - **os.X_OK** 包含在access()的mode参数中 ，测试path是否可执行。
2. os.chdir(path): 用于改变当前工作目录到指定的路径。如果允许访问返回`True`, 否则返回`False`。
3. os.chmod(path, mode): 用于更改文件或目录的权限。
    - **flags** -- 可用以下选项按位或操作生成，目录的读权限表示可以获取目录里的文件名列表，执行权限表示可以把工作目录切换到此目录。删除添加目录里的文件必须同时有写和执行权限。文件权限以 "用户 ID ->组 ID -> 其它" 顺序检验，最先匹配的允许或禁止权限被应用。
        - **stat.S_IXOTH:** 其他用户有执行权0o001
        - **stat.S_IWOTH:** 其他用户有写权限0o002
        - **stat.S_IROTH:** 其他用户有读权限0o004
        - **stat.S_IRWXO:** 其他用户有全部权限0o007
        - **stat.S_IXGRP:** 组用户有执行权限0o010
        - **stat.S_IWGRP:** 组用户有写权限0o020
        - **stat.S_IRGRP:** 组用户有读权限0o040
        - **stat.S_IRWXG:** 组用户有全部权限(权限掩码)0o070
        - **stat.S_IXUSR:** 拥有者具有执行权限0o100
        - **stat.S_IWUSR:** 拥有者具有写权限0o200
        - **stat.S_IRUSR:** 拥有者具有读权限0o400
        - **stat.S_IRWXU:** 拥有者有全部权限(权限掩码)0o700
        - **stat.S_ISVTX:** 目录里的文件目录拥有者才可删除更改0o1000
        - **stat.S_ISGID:** 执行此文件其进程有效组为文件所在组0o2000
        - **stat.S_ISUID:** 执行此文件其进程有效用户为其所有者0o4000
        - **stat.S_IREAD:** windows下设为只读
        - **stat.S_IWRITE:** windows下取消只读
4. os.chown(path, uid, gid): 更改文件所有者。需要超级用户权限，只支持在 Unix 下使用。
5. os.chroot(path): 改变当前进程的根目录。需要超级用户权限，只支持在 Unix 下使用。
6. os.close(fd): 关闭指定的文件描述符`fd`。
7. os.closerange(fd_low, fd_high): 关闭从`fd_low`(包含)到`fd_high`(不包含)的文件描述符, 错误会忽略。
8. **cpu_count()**: 返回系统的CPU数。若想得知当前进程的可用CPU数，则用`len(os.sched_getaffinity(0))`。
9. os.dup(fd): 复制文件描述符`fd`，返回被复制的文件描述符。
10. os.dup2(fd1, fd2): 将一个文件描述符`fd1`复制到另一个`fd2`。e.g. `os.dup2(f.fileno(), 1) # 1 -> stdout`
11. os.fchdir(fd): 通过文件描述符改变当前工作目录。

```python
#!/usr/bin/env python3
import os, sys
os.chdir("/var/www/html" )  # 切换到 "/var/www/html" 目录
print ("当前工作目录 : %s" % os.getcwd())  # 打印当前目录
fd = os.open( "/tmp", os.O_RDONLY )  # 打开 "/tmp"
os.fchdir(fd)  # 使用 os.fchdir() 方法修改目录
print ("当前工作目录 : %s" % os.getcwd())  # 打印当前目录
os.close(fd)  # 关闭文件
```

12. os.fchmod(fd, mode): 用于改变一个由`fd`指定的文件的访问权限，参数`mode`是Unix下的文件访问权限。

13. **os.fchown(fd, uid, gid)**: 用于修改一个由`fd`指定的文件的所有权，即文件的用户ID和用户组ID。
14. os.fdatasync(fd): 强制将一个由`fd`指定的文件写入磁盘但不强制更新文件的状态信息。可用于刷新缓冲区。
15. **os.fdopen(fd[, mode[, bufsize]])**: 通过文件描述符`fd`创建一个文件对象，并返回这个文件对象。

    实例：`fd = os.open("foo.txt", os.O_RDWR|os.O_CREAT); fo = os.fdopen(fd, "w+"); fo.write("Ha\n")`
16. **os.fstat(fd)**: 返回文模式件描述符`fd`的状态，该方法返回的结构：
    - **st_dev:** 设备信息
    - **st_ino:** 文件的`i-node`值
    - **st_mode:** 文件信息的掩码，包含了文件的权限信息，文件的类型信息(普通文件还是管道文件或是其他的文件类型)
    - **st_nlink:** 硬连接数
    - **st_uid:** 用户ID
    - **st_gid:** 用户组 ID
    - **st_rdev:** 设备 ID (如果指定文件)
    - **st_size:** 文件大小，以`byte`为单位
    - **st_blksize:** 系统 I/O 块大小
    - **st_blocks:** 文件是由多少个 512`byte`的块构成的
    - **st_atime:** 文件最近的访问时间
    - **st_mtime:** 文件最近的修改时间
    - **st_ctime:** 文件状态信息的修改时间 (而非文件内容的修改时间)
17. os.fsync(fd): 强制将文件描述符为`fd`的文件写入硬盘。
18. os.ftruncate(fd, length): 裁剪文件描述符`fd`对应的文件，它最大不能超过文件大小。
19. os.fork(): Fork a child process. Return 0 to child process & PID of child to parent process.
20. **os.getcwd()**: 用于返回当前工作目录。
21. **os.getenv(str)**: 用于返回指定系统变量的路径。
22. **os.link(src, dst)**: 用于创建从目标地址`dst`指向源地址`src`的硬链接。
23. **os.listdir(path)**: 返回指定文件夹所包含的文件或文件夹名字列表，以字母顺序排序。
24. **os.lseek(fd, offset, from_what)**: 用于设置文件描述符`fd`的当前位置。
25. **os.makedirs(path[, mode])**: 用于递归创建目录。可选参数`mode`为权限模式。实例: `import os,sys; path = "/tmp/home/monthly/daily"; os.makedirs(path, 0o777)`
26. **os.mkdir(path[, mode])**: 用于以数字权限模式创建目录。
27. **os.open(file, flags[, mode=511])**: 用于打开一个文件并且设置需要的打开选项。
28. os.pipe(): 用于创建一个管道, 返回一对文件描述符 (read_fd, write_fd) 分别为读和写。
29. **os.read(fd, n)**: 从文件描述符`fd`中读取最多`n`个字节，返回读取字节的字符串。若文件描述符`fd`对应文件已达到结尾，则返回一个空字符串。
30. os.readlink(path): 用于返回软链接所指向的文件，可能返回绝对或相对路径。
31. **os.remove(path)**: 用于**删除指定**路径的**文件**。如果指定的路径是一个目录，将抛出`OSError`。
32. **os.removedirs(path)**: 像`rmdir()`，但用于递归删除目录。若子文件夹成功删除，才尝试父文件夹。
33. **os.rename(src, dst)**: 用于重命名文件或目录，从`src`到`dst`。若`dst`是一个存在的目录将抛出`OSError`。
34. **os.renames(old, new)**: 用于递归重命名目录或文件。类似`rename()`。

35. **os.replace(src, dst)**: Rename a file or directory, overwriting the destination.
36. **os.rmdir(path)**: 用于删除指定路径的目录，仅当该文件夹是空的才可， 否则抛出`OSError`。
37. os.stat(path): 获取`path`指定的路径的信息。返回的`stat`的结构：
    - **st_mode:** inode 保护模式
    - **st_ino:** inode 节点号。
    - **st_dev:** inode 驻留的设备。
    - **st_nlink:** inode 的链接数。
    - **st_uid:** 所有者的用户ID。
    - **st_gid:** 所有者的组ID。
    - **st_size:** 普通文件以字节为单位的大小；包含等待某些特殊文件的数据。
    - **st_atime:** 上次访问的时间。
    - **st_mtime:** 最后一次修改的时间。
    - **st_ctime:** 某些系统上(如Unix)是最新的元数据更改的时间，在其它系统上(如Windows)是创建时间。
38. **os.symlink(src, dst)**: 用于创建从目标地址`dst`指向源地址`src`的软链接。
39. **os.system(str_command)**: Execute the command in a subshell.
40. os.tcgetpgrp(fd): 用于返回与终端`fd` (一个由`os.open()`返回的打开的文件描述符) 关联的进程组。
41. os.tcsetpgrp(fd, pg): 用于设置与终端`fd` (一个由`os.open()`返回的打开的文件描述符) 关联的进程组为`pg`。
42. os.unlink(path): 用于删除文件，若文件是一个目录则返回一个错误。
43. os.utime(path, times): 用于设置指定路径文件最后的修改和访问时间。若`times`是`None`，则文件的访问和修改设为当前时间 。否则`times`是一个`tuple`数字`(atime, mtime)`用来分别作为访问和修改的时间。
44. **os.times()**: 返回一个元组: (utime, stime, cutime, cstime, elapsed_time), 所有时间都为浮点数。
45. **os.walk(top[, topdown=True[, onerror=None[, followlinks=False]])**: 可以创建一个生成器，用以生成所要查找的目录及其子目录下的所有文件。用于通过在目录树中向上或向下游走输出在目录中的文件名。是一个简单易用的文件及目录的遍历器，可以帮助高效地处理文件及目录。
    - **top** -- 根目录下的每一个文件夹(包含它自己), 产生一个3-元组`(dirpath, dirnames, filenames)`即 (文件夹路径, 文件夹名字, 文件名)。
    - **topdown** -- 可选。若为`True`或没有指定时，一个目录的的3-元组将比它的任何子文件夹的3-元组先产生 (即目录自上而下)。若为`False`，一个目录的3-元组将比它的任何子文件夹的3-元组后产生 (即自下而上)。
    - **onerror** -- 可选。是一个函数，它调用时有一个参数，一个`OSError`实例。报告这错误后，继续walk或者抛出`exception`终止walk。
    - **followlinks** -- 若设置为`True`则通过软链接访问目录。

    ```python
    #!/usr/bin/env python3
    import os
    for root, dirs, files in os.walk(".", topdown=True):
        for name in files:
            print(os.path.join(root, name))
        for name in dirs:
            print(os.path.join(root, name))
    ```

46. os.fwalk(top='.', topdown=True, onerror=None, *, follow_symlinks=False, dir_fd=None): 与`os.walk`类似，区别在于该函数返回4-元组`(dirpath, dirnames, filenames, dirfd)`。

47. **os.write(fd, str)**: 用于写入字符串到文件描述符`fd`中，且返回实际写入的字符串长度。
48. **os.path 模块**: 主要用于获取文件的属性。

Method	Description

Method	Description
`os.path.abspath(path)`	返回绝对路径
`os.path.basename(path)`	返回文件名
`os.path.commonprefix(list)`	返回`list`(多个路径)中所有`path`共有的最长的路径
`os.path.dirname(path)`	返回文件路径
`os.path.exists(path)`	路径存在则返回`True`,路径损坏返回`False`
`os.path.lexists`	路径存在则返回`True`,路径损坏也返回`True`
`os.path.expanduser(path)`	把`path`中包含的"~"和"~user"转换成用户目录
`os.path.expandvars(path)`	根据环境变量的值替换`path`中包含的"$name"和"${name}"
`os.path.getatime(path)`	返回最近访问时间(浮点型秒数)
`os.path.getmtime(path)`	返回最近文件修改时间
`os.path.getctime(path)`	返回文件`path`创建时间
`os.path.getsize(path)`	返回文件大小，若文件不存在则返回错误
`os.path.isabs(path)`	判断是否为绝对路径
`os.path.isfile(path)`	判断路径是否为文件
`os.path.isdir(path)`	判断路径是否为目录
`os.path.islink(path)`	判断路径是否为链接
`os.path.ismount(path)`	判断路径是否为挂载点
`os.path.join(path1[, path2[, ...]])`	把目录和文件名合成一个路径
`os.path.normcase(path)`	转换`path`的大小写和斜杠
`os.path.normpath(path)`	规范`path`字符串形式
`os.path.realpath(path)`	返回`path`的真实路径
`os.path.relpath(path[, start])`	从`start`开始计算相对路径
`os.path.samefile(path1, path2)`	判断目录或文件是否相同
`os.path.sameopenfile(fp1, fp2)`	判断`fp1`和`fp2`是否指向同一文件
`os.path.samestat(stat1, stat2)`	判断stat元组`stat1`和`stat2`是否指向同一个文件
`os.path.split(path)`	把路径分割成`dirname`和`basename`并返回一个元组

Method	Description
`os.path.splitdrive(path)`	一般用在Windows下返回驱动器名和路径组成的元组
`os.path.splitext(path)`	分割路径，返回路径名和文件扩展名的元组
`os.path.splitunc(path)`	把路径分割为加载点与文件
`os.path.walk(path, visit, arg)`	进入`path`每个目录都调用`visit`函数, 必须有3个参数: (`arg, dirname, names`)，其中`dirname`表示当前目录名, `names`表示当前目录下所有文件名; `args`则为walk的第三个参数
`os.path.supports_unicode_filenames`	设置是否支持Unicode路径名

实例1：

```
#!/usr/bin/env python3
import os
print( os.path.basename('/root/runoob.txt') )    # 返回文件名
print( os.path.dirname('/root/runoob.txt') )     # 返回目录路径
print( os.path.split('/root/runoob.txt') )       # 分割文件名与路径
print( os.path.join('root','test','runoob.txt') )  # 将目录和文件名合成一个路径
```

实例2：

```
#!/usr/bin/env python3
import os, time
file='./runoob.txt'  # 文件路径
print( os.path.getatime(file) )   # 输出最近访问时间
print( os.path.getctime(file) )   # 输出文件创建时间
print( os.path.getmtime(file) )   # 输出最近修改时间
print( time.gmtime(os.path.getmtime(file)) )  # 以struct_time形式输出最近修改时间
print( os.path.getsize(file) )   # 输出文件大小（字节为单位）
print( os.path.abspath(file) )   # 输出绝对路径
print( os.path.normpath(file) )  # 规范path字符串形式
```

实例3：(获取指定目录及其子目录下的".py"文件)

```
#!/usr/bin/env python3
import os
import os.path

ls = []  # save full paths of all '.py' files found

def get_appoint_file(path, ls):  # path: full path specified
    file_list = os.listdir(path)  # list all file names in the specified path
    try:
        for tmp in file_list:  # iterate for all file names
            path_tmp = os.path.join(path, tmp)  # assemble full path of one file
            if os.path.isdir(path_tmp):  # if the file is still a folder
                get_appoint_file(path_tmp, ls)  # invoke the subroutine recursively
            elif path_tmp[path_tmp.rfind('.') + 1:].upper() == 'PY':  # if suffix
is '.py'
```

```
                ls.append(path_tmp)  # found one python file
    except PermissionError:  # if there is no permission to visit the file
        pass  # continue to search for next file

def main():
    while True:
        path = input('Please type an absolute location:').strip()  # remove
whitespaces
        if os.path.isdir(path):
            break

    get_appoint_file(path, ls)
    print('In the folder %s found %d py files: ' % (path, len(ls)))
    for filepath in ls:
        print('  ' + filepath)

if __name__ == '__main__':
    main()
```

实例4：(显示指定目录下的.mp4, .avi, .rmvb视频格式文件)

```
import os

def search_file(start_dir, target) :
    os.chdir(start_dir) # 切换当前工作路径至指定目录
    for each_file in os.listdir(os.curdir) :
        ext = os.path.splitext(each_file)[1]
        if ext in target :
            vedio_list.append(os.getcwd() + os.sep + each_file + os.linesep)
        if os.path.isdir(each_file) :
            search_file(each_file, target) # 递归调用
            os.chdir(os.pardir) # 递归调用后切记返回上一层目录

start_dir = input('请输入待查找的初始目录：')
program_dir = os.getcwd()

target = ['.mp4', '.avi', '.rmvb']
video_list = []

search_file(start_dir, target)

f = open(program_dir + os.sep + 'VideoList.txt', 'w')
f.writelines(video_list)
f.close()
```

实例5：(搜索并替换一指定文本文件中的字符或单词)

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

def file_replace(file_name, rep_word, new_word):
    f_read = open(file_name)

    content = []
    count = 0
    for eachline in f_read:
        if rep_word in eachline:
```

```
            count = count + eachline.count(rep_word)
            eachline = eachline.replace(rep_word, new_word)
        content.append(eachline)

    message = '\nFile %s contains %s [%s]' \
              '\nDo you what to replace all [%s] by [%s]?' \
              '\n[YES/NO]:'
    decide = input(message % (file_name, count, rep_word, rep_word, new_word))

    if decide in ['YES', 'Yes', 'yes']:
        f_write = open(file_name, 'w')
        f_write.writelines(content)
        f_write.close()

    f_read.close()

if __name__ == "__main__":
    file_name = input('Please Type a Text File:')
    rep_word = input('Please Type a Character/Word to be Replaced:')
    new_word = input('Please Type a New Character/Word:')
    file_replace(file_name, rep_word, new_word)
```

3 Python 3 标准库概览

3.1 操作系统接口

Python 中`os`模块提供了不少与操作系统相关联的函数。

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

3.2 使用`shutil`模块

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

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

3.3 文件通配符

Python 中的`glob`模块提供了一个函数用于从目录通配符搜索中生成文件列表:

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

3.4 命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于`sys`模块的`argv`变量。例如在命令行中执行`python demo.py one two three`后可以得到以下输出结果:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

3.5 错误输出重定向和程序终止

Python 中的`sys`模块还有`stdin`，`stdout`和`stderr`属性，即使在`stdout`被重定向时，后者也可以用于显示警告和错误信息。大多脚本的定向终止都使用`sys.exit()`。

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

3.6 字符串正则匹配

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

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

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

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

3.7 数学库

Python 中的`math`模块为浮点运算提供了对底层C函数库的访问：

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

Python 中的`random`提供了生成随机数的工具：

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

3.8 访问互联网

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

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8')  # Decoding the binary data to text.
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

3.9 日期和时间

Python 中的`datetime`模块为日期和时间处理同时提供了简单和复杂的方法。支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。该模块还支持时区处理:

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

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

常用时间处理方法

- 今天 `today = datetime.date.today()`
- 昨天 `yesterday = today - datetime.timedelta(days=1)`
- 上个月 `last_month = today.month - 1 if today.month - 1 else 12`
- 当前时间戳 `time_stamp = time.time()`
- 时间戳转datetime `datetime.datetime.fromtimestamp(time_stamp)`
- datetime转时间戳 `int(time.mktime(today.timetuple()))`
- datetime转字符串 `today_str = today.strftime("%Y-%m-%d")`
- 字符串转datetime `today = datetime.datetime.strptime(today_str, "%Y-%m-%d")`
- 补时差 `today + datetime.timedelta(hours=8)`

3.10 数据压缩

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

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

3.11 性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。例如：使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多，`timeit`证明了现代的方法更快一些。

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

相对于`timeit`的细腻度，`profile`和`pstats`模块提供了针对更大代码块的时间度量工具。

3.12 测试模块

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试。Python 中`doctest`模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。通过用户提供的例子强化了文档，允许`doctest`模块确认代码的结果是否与文档一致：

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

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

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

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

```python
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

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

4 使用`shutil`模块操作文件

Python 中的`shutil`模块是高级的文件、文件夹、压缩包处理模块。

1. **shutil.copyfileobj(fsrc, fdst[, length])**: 将`fsrc`文件的数据覆盖拷贝给`fdst`文件

   ```python
   import shutil
   f1 = open("1.txt","r",encoding="utf-8")
   f2 = open("2.txt","w",encoding="utf-8")
   shutil.copyfileobj(f1,f2)
   ```

2. **shutil.copyfile(src, dst)**: 不用打开文件，直接用文件名进行覆盖拷贝

   ```python
   import shutil
   shutil.copyfile("1.txt","3.txt")
   ```

3. **shutil.copy(src, dst)**: 拷贝文件和权限, `e.g. shutil.copy('f1.log', 'f2.log')`
4. **shutil.copy2(src, dst)**: 拷贝文件和状态信息, e.g. `shutil.copy2('f1.log', 'f2.log')`
5. **shutil.copymode(src, dst)**: 只拷贝权限，内容组,用户组均不变 (`dst`文件必须存在，否则报错)

   ```python
   shutil.copymode('f1.log', 'f2.log')
   ```

6. **shutil.copystat(src, dst)**: 仅拷贝文件的状态信息(文件属性)，包括: mode bits, atime, mtime, flags

7. **shutil.ignore_patterns(*patterns)**: 忽略模式，用于配合`copytree()`方法，传递会被忽略不被拷贝的文件
8. **shutil.copytree(src, dst, symlinks=False, ignore=None)**: 递归的去拷贝文件夹

```
shutil.copytree('folder1', 'folder2', ignore=shutil.ignore_patterns('*.pyc', 'tmp*'))
shutil.copytree('f1', 'f2', symlinks=True, ignore=shutil.ignore_patterns('*.pyc', 'tmp*'))

from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

9. **shutil.rmtree(path[, ignore_errors=False[, onerror=None]])**: 递归删除指定目录下的目录及文件
10. **shutil.move(src, dst)**: 递归的去移动文件，类似`mv`命令，其实就是重命名
11. **shutil.disk_usage(path)**: 获取文件路径或文件夹路径所在硬盘使用情况

```
import shutil, os
path = os.path.join(os.getcwd(),"aaa")
info = shutil.disk_usage(path)
print(info)   # usage(total=95089164288, used=7953104896, free=87136059392)
```

12. **shutil.chown(path, user=None, group=None)**: 修改路径指向的文件或文件夹的所有者或分组。

```
import shutil,os
path = os.path.join(os.getcwd(),"file.txt")
shutil.chown(path,user="root",group="root")
```

13. **shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)**: 获取给定的cmd命令的可执行文件的路径。

```
import shutil
info = shutil.which("python3")
print(info)   # /usr/bin/python3
```

14. **shutil.make_archive(base_name, format,...)**: 创建压缩包并返回文件路径，如: zip, tar, gztar 等

- - base_name : 压缩包的文件名或路径。只是文件名时，则保存至当前目录；否则保存至指定路径。
    如: www => 保存至当前路径
    如: /Users/wupeiqi/www => 保存至/Users/wupeiqi/中
  - format : 压缩包种类，支持 "zip", "tar", "bztar", "gztar"
  - root_dir : 要压缩的文件夹路径 (默认当前目录)
  - owner : 用户 (默认当前用户)

    - group : 组 (默认当前组)
    - logger : 用于记录日志 (通常是`logging.Logger`对象)

```
>>> import shutil
# 将 /Downloads/test 下的文件打包放置当前程序目录下的 test.tar.gz 中
>>> ret = shutil.make_archive("test1", 'gztar', root_dir='/Downloads/test')
# 将 /Downloads/test 下的文件打包放置 /usr/ 目录下的 test.tar.gz 中
>>> ret = shutil.make_archive("/usr/test2", 'gztar',
root_dir='/Downloads/test')
>>>
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

15. **shutil.get_archive_formats()**: 获取支持的压缩文件格式。目前支持"zip", "tar", "bztar", "gztar", " xztar "
16. **shutil.unpack_archive(filename, extract_dir=None, format=None)**: 解压操作
    - filename : 压缩包的文件路径
    - extract_dir : 解压至的文件夹路径 (文件夹可以不存在, 可自动生成)
    - format :解压格式，默认为`None`，会根据扩展名自动选择解压格式
17. **shutil.get_unpack_formats()**: 获取支持的解压文件格式。目前支持"zip", "tar", "bztar", "gztar", " xztar "
18. `shutil`对压缩包的处理是主要依靠的是调用`ZipFile`和`TarFile`两个模块来进行的，详细：

```
import zipfile
# 压缩
z = zipfile.ZipFile('laxi.zip', 'w')
z.write('a.log')
z.write('data.data')
z.close()
# 解压
z = zipfile.ZipFile('laxi.zip', 'r')
z.extractall()
z.close()

import tarfile
# 压缩
tar = tarfile.open('your.tar','w')
tar.add('/Users/wupeiqi/PycharmProjects/bbs2.zip', arcname='bbs2.zip')
tar.add('/Users/wupeiqi/PycharmProjects/cmdb.zip', arcname='cmdb.zip')
tar.close()
# 解压
tar = tarfile.open('your.tar','r')
tar.extractall()  # 可设置解压地址
tar.close()
```

- 备注：`zipfile`**压缩不会保留文件的状态信息**，而`tarfile`**会保留文件的状态信息**。

5 Python 获取命令行参数

5.1 利用`sys.argv`

Python 中可以用 `sys` 的 `sys.argv` 来获取命令行参数：

```
sys.argv 是命令行参数列表。
len(sys.argv) 是命令行参数个数

注：sys.argv[0] 表示代码本身文件路径，所以参数从1开始
```

5.1.1 实例1

创建test.py 文件，代码如下：

```
#!/usr/bin/env python3
import sys
print ('参数个数为:', len(sys.argv), '个参数。')
print ('参数列表:', str(sys.argv))
```

执行以上代码，输出结果为：

```
$ python3 test.py arg1 arg2 arg3
参数个数为: 4 个参数。
参数列表: ['test.py', 'arg1', 'arg2', 'arg3']
```

5.1.2 实例2

创建sample.py 文件，代码如下：

```
#!/usr/bin/env python
#_*_ coding:utf-8 _*_
import sys

HELP = '''
This program prints files to the standard output.
Any number of files can be specified.
Options include:
  --version : Prints the version number
  --help    : Display this help
'''

def readfile(filename): #定义readfile函数，从文件中读出文件内容
  '''''''''Print a file to the standard output.'''
  f = file(filename)
  while True:
    line = f.readline()
    if len(line) == 0:
      break
    print line, # notice comma 分别输出每行内容
  f.close()

# Script starts from here
print sys.argv

if len(sys.argv) < 2:
```

```
  print 'No action specified.'
  sys.exit()

if sys.argv[1].startswith('--'):
  option = sys.argv[1][2:]
  # fetch sys.argv[1] but without the first two characters
  if option == 'version': #当命令行参数为-- version，显示版本号
    print 'Version 1.2'
  elif option == 'help': #当命令行参数为--help时，显示相关帮助内容
    print HELP
  else:
    print 'Unknown option.'
  sys.exit()
else:
  for filename in sys.argv[1:]: #当参数为文件名时，传入readfile，读出其内容
    readfile(filename)
```

在与sample.py同一目录下，新建1个记事本文件test.txt, 其内容为: `hello python!`。
验证sample.py，如下：

```
C:\Users\91135\Desktop>python sample.py
 ['sample.py']
No action specified.

C:\Users\91135\Desktop>python sample.py --help
['sample.py', '--help']
This program prints files to the standard output.
 Any number of files can be specified.
 Options include:
  --version : Prints the version number
  --help    : Display this help

C:\Users\91135\Desktop>python sample.py --version
 ['sample.py', '--version']
Version 1.2

C:\Users\91135\Desktop>python sample.py --ok
 ['sample.py', '--ok']
Unknown option.

C:\Users\91135\Desktop>python sample.py test.txt
 ['sample.py', 'test.txt']
hello python!
```

5.2 利用`getopt`模块

`getopt`模块是专门处理命令行参数的模块，用于获取命令行选项和参数，即`sys.argv`。命令行选项使得程序的参数更加灵活。支持短选项模式 (-) 和长选项模式 (--)。该模块提供了两个方法及一个异常处理来解析命令行参数。

5.2.1 `getopt.getopt`方法

`getopt.getopt`方法用于解析命令行参数列表，语法格式如下：

```
getopt.getopt(args, options[, long_options])
```

方法参数说明:

```
args         : 要解析的命令行参数列表。
options      : 以字符串的格式定义，后的冒号(:)表示该选项必须有附加参数，不带冒号表示该选项不附加参数。
long_options: 以列表的格式定义，后的等号(=)表示如果设置该选项则必须有附加参数，否则就不附加参数。

该方法返回值由两个元素组成:
第一个是 (option, value) 元组的列表。
第二个是参数列表，包含那些没有'-'或'--'的参数。
```

5.2.2 `getopt.gnu_getopt` 方法

另外一个方法是 'getopt.gnu_getopt'，这里不多做介绍。

5.2.3 `except getopt.GetoptError`

在没有找到参数列表，或选项的需要的参数为空时会触发该异常。
异常的参数是一个字符串，表示错误的原因。
属性 `msg` 和 `opt` 为相关选项的错误信息。

5.2.4 实例

假定我们创建这样一个脚本，可以通过命令行向脚本文件传递两个文件名，同时我们通过另外一个选项查看脚本的使用。脚本使用方法如下：

```
usage: test.py -i <inputfile> -o <outputfile>
```

创建test.py 文件，代码如下所示：

```
#!/usr/bin/env python3
import sys, getopt

def main(argv):
    inputfile = ''
    outputfile = ''
    try:
        opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
    except getopt.GetoptError:
        print ('test.py -i <inputfile> -o <outputfile>')
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print ('test.py -i <inputfile> -o <outputfile>')
            sys.exit()
        elif opt in ("-i", "--ifile"):
            inputfile = arg
        elif opt in ("-o", "--ofile"):
            outputfile = arg
        print ('Input File is: ', inputfile)
        print ('Output File is: ', outputfile)
```

```
22
23 if __name__ == "__main__":
24     main(sys.argv[1:])
```

```
1 Note:
2     "hi:o:"             -> '-'型参数有: -h, -i(必须带附加参数), -o(必须带附加参数)
3     ["ifile=","ofile="] -> '--'型参数有: --ifile(必须带附加参数), --ofile(必须带附加参数)
```

执行以上代码，输出结果为：

```
1 $ python3 test.py -h
2 usage: test.py -i <inputfile> -o <outputfile>
3 $ python3 test.py -i inputfile -o outputfile
4 Input File is: inputfile
5 Output File is: outputfile
```

6 正则表达式

正则表达式是一个特殊的字符序列，能帮助方便地检查一个字符串是否与某种模式匹配。Python 自1.5版本起增加了`re`模块，提供 Perl 风格的正则表达式模式，使 Python 语言拥有全部的正则表达式功能。此外，`compile`函数可根据一个模式字符串和可选的标志参数生成一个正则表达式对象，该对象拥有一系列方法用于正则表达式匹配和替换。`re`模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串作为它们的第一个参数。

6.1 使用`re`模块的方法

6.1.1 `re.match`方法

`re.match`尝试**从字符串的起始位置匹配**一个模式，匹配成功返回一个匹配的对象，否则返回`None`。

函数语法：`re.match(pattern, string, flags=0)`

参数：

- pattern : 匹配的正则表达式。
- string : 要匹配的字符串。
- flags : 可选，表示匹配模式，比如忽略大小写，多行模式等。具体参数为：
    - **re.I** : 忽略大小写
    - **re.L** : 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境
    - **re.M** : 多行匹配，影响 ^ 和 $
    - **re.S** : 即为' . '并且包括换行符在内的任意字符（注：' . '不包括换行符）
    - **re.U** : 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库。
    - **re.X** : 该标志给予更灵活的格式，忽略空格和' # '后面的注释以增加可读性。

可使用`group(num)`或`groups()`匹配对象函数来获取匹配表达式：

- `group(num=0)`: 匹配的整个表达式的字符串，可一次输入多个组号，则返回包含那些组所对应值的元组。
- `groups()`: 返回一个包含所有小组字符串的元组，从 1 到所含的组号。

实例1：

```
>>> import re
>>> print(re.match('www', 'www.runoob.com').span())  # 在起始位置匹配
(0, 3)
>>> print(re.match('com', 'www.runoob.com'))         # 不在起始位置匹配
None
```

实例2：

```
#!/usr/bin/env python3
import re
line = "Cats are smarter than dogs"
# .* 表示任意匹配除换行符(/n,/r)之外的任何单个或多个字符
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
   print ("matchObj.group() : ", matchObj.group())
   print ("matchObj.group(1) : ", matchObj.group(1))
   print ("matchObj.group(2) : ", matchObj.group(2))
else:
   print ("No match!!")
```

输出结果：

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

6.1.2 `re.search`方法

`re.search`扫描整个字符串并返回第一个成功的匹配。若匹配成功返回一个匹配的对象，否则返回`None`。

函数语法：`re.search(pattern, string, flags=0)`

参数：

- pattern : 匹配的正则表达式。
- string : 要匹配的字符串。
- flags : 可选，表示匹配模式，比如忽略大小写，多行模式等。

同样可以使用`group(num)`或`groups()`匹配对象函数来获取匹配表达式。

```
>>> import re
>>> print(re.match('www', 'www.runoob.com').span())  # 在起始位置匹配
(0, 3)
>>> print(re.match('com', 'www.runoob.com').span())  # 不在起始位置匹配
(11, 14)
```

`re.match`只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败返回`None`；而`re.search`匹配整个字符串，直到找到一个匹配，则匹配失败返回`None`。

```
#!/usr/bin/env python3
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
```

```
   print ("match --> matchObj.group() : ", matchObj.group())
else:
   print ("No match!!")

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
   print ("search --> matchObj.group() : ", matchObj.group())
else:
   print ("No match!!")
```

运行结果：

```
No match!!
search --> matchObj.group() :  dogs
```

6.1.3 检索和替换

Python 的`re`模块提供了`re.sub`用于替换字符串中的匹配项。

函数语法：`re.sub(pattern, repl, string[, count=0[, flags=0]])`

参数：(前三个为必选参数，后两个为可选参数)

- pattern : 正则中的模式字符串。
- repl : 替换的字符串，也**可为一个函数**。
- string : 要被查找替换的原始字符串。
- count : 模式匹配后替换的最大次数，默认0表示替换所有的匹配。
- flags : 编译时用的匹配模式，数字形式。

```
#!/usr/bin/env python3
import re
phone = "2004-959-559 # 这是一个电话号码"

num = re.sub(r'#.*$', "", phone)  # 删除注释
print ("电话号码 : ", num)

num = re.sub(r'\D', "", phone)  # 移除非数字的内容
print ("电话号码 : ", num)
```

运行结果：

```
电话号码 :  2004-959-559
电话号码 :  2004959559
```

`repl`参数可为一个函数:

```python
#!/usr/bin/env python3
import re

def double(matched):  # 将匹配的数字乘r2
    value = int(matched.group('value'))
    return str(value * 2)

s = 'A23G4HFD567'
print(re.sub('(?P<value>\d+)', double, s))

# 输出结果: A46G8HFD1134
```

6.1.4 `re.compile`方法

`compile`函数用于编译正则表达式，**生成一个正则表达式(`Pattern`)**供`match()`和`search()`这两个函数使用。

函数语法：`re.compile(pattern[, flags])`

参数：

- pattern : 一个字符串形式的正则表达式
- flags : 可选，表示匹配模式，比如忽略大小写，多行模式等。

实例1：

```python
>>> import re
>>> pattern = re.compile(r'\d+')                    # 用于匹配至少一个数字
>>> m = pattern.match('one12twothree34four')        # 查找头部，没有匹配
>>> print(m)
None
>>> m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配
>>> print(m)
None
>>> m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配，正好匹配
>>> print(m)                                        # 返回一个 Match 对象
<re.Match object; span=(3, 5), match='12'>
>>> m.group(0)   # 可省略 0
'12'
>>> m.start(0)   # 可省略 0
3
>>> m.end(0)     # 可省略 0
5
>>> m.span(0)    # 可省略 0
(3, 5)
```

在上面，当匹配成功时返回一个 Match 对象，其中：

- **`group([group1, …])`**方法用于获得一个或多个分组匹配的字符串。当要获得整个匹配的子串时，可直接使用`group()`或`group(0)`；
- **`start([group])`**方法用于获取分组匹配的子串在整个字符串中的起始位置 (子串第一个字符的索引)，参数默认值为 0；
- **`end([group])`**方法用于获取分组匹配的子串在整个字符串中的结束位置 (子串最后一个字符的索引+1)，参数默认值为 0；
- **`span([group])`**方法返回`(start(group), end(group))`。

实例2：

```
>>> import re
>>> pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)   # re.I 表示忽略大小写
>>> m = pattern.match('Hello World Wide Web')
>>> print m                               # 匹配成功，返回一个 Match 对象
<re.Match object; span=(0, 11), match='Hello World'>
>>> m.group(0)                            # 返回匹配成功的整个子串
'Hello World'
>>> m.span(0)                             # 返回匹配成功的整个子串的索引
(0, 11)
>>> m.group(1)                            # 返回第一个分组匹配成功的子串
'Hello'
>>> m.span(1)                             # 返回第一个分组匹配成功的子串的索引
(0, 5)
>>> m.group(2)                            # 返回第二个分组匹配成功的子串
'World'
>>> m.span(2)                             # 返回第二个分组匹配成功的子串索引
(6, 11)
>>> m.groups()                            # 等价于 (m.group(1), m.group(2), ...)
('Hello', 'World')
>>> m.group(3)                            # 不存在第三个分组
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

6.1.5 `re.findall`方法

在字符串中找到正则表达式所匹配的所有子串，并返回一个列表。若没有找到匹配，则返回空列表。

注意： `re.match`和`re.search`是匹配一次，而`re.findall`匹配所有。

函数语法：`re.findall(string[, pos[, endpos]])`

参数：

- string : 待匹配的字符串。
- pos : 可选参数，指定字符串的起始位置，默认为 0。
- endpos : 可选参数，指定字符串的结束位置，默认为字符串的长度。

查找字符串中的所有数字：

```
>>> import re
>>> pattern = re.compile(r'\d+')  # 查找数字
>>> result1 = pattern.findall('runoob 123 google 456')
>>> result2 = pattern.findall('run88oob123google456', 0, 10)
>>> print(result1)
['123', '456']
>>> print(result2)
['88', '12']
```

6.1.6 `re.finditer`方法

和`re.findall`类似，在字符串中找到正则表达式所匹配的所有子串，并把它们作为一个迭代器返回。

函数语法：`re.finditer(pattern, string, flags=0)`

参数：

- pattern : 匹配的正则表达式。
- string : 要匹配的字符串。
- flags : 可选，表示匹配模式，比如忽略大小写，多行模式等。

```
>>> import re
>>> it = re.finditer(r"\d+","12a32bc43jf3")
>>> for match in it:
...   print(match.group())
...
12
32
43
3
```

6.1.7 `re.split`方法

`re.split`方法按照能够匹配的子串将字符串分割后返回列表。

函数语法：`re.split(pattern, string[, maxsplit=0, flags=0])`

参数：

- pattern : 匹配的正则表达式。
- string : 要匹配的字符串。
- maxsplit : 分隔次数，若`maxsplit=1`则分隔一次，默认为 0，不限制次数。
- flags : 可选，表示匹配模式，比如忽略大小写，多行模式等。

```
>>>import re
>>> re.split('\W+', 'runoob, runoob, runoob.')
['runoob', 'runoob', 'runoob', '']
>>> re.split('(\W+)', ' runoob, runoob, runoob.')
['', ' ', 'runoob', ', ', 'runoob', ', ', 'runoob', '.', '']
>>> re.split('\W+', ' runoob, runoob, runoob.', 1)
['', 'runoob, runoob, runoob.']
>>> re.split('a*', 'hello world')  # 对于一个找不到匹配的字符串而言，re.split不会对其作出分割
['hello world']
```

6.2 正则表达式对象

6.2.1 `re.RegexObject`

`re.compile()`返回`RegexObject`对象。

6.2.2 `re.MatchObject`

`group()`返回的被`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

6.3 正则表达式修饰符

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位`OR`或`|`们来指定。如：`re.I | re.M` 被设置成 I 和 M 标志。

- **re.I** : 忽略大小写
- **re.L** : 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境
- **re.M** : 多行匹配，影响 ^ 和 $
- **re.S** : 即为' . '并且包括换行符在内的任意字符（注： ' . '不包括换行符）
- **re.U** : 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库。
- **re.X** : 该标志给予更灵活的格式，忽略空格和' # '后面的注释以增加可读性。

6.4 正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

- 字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。
- 多数字母和数字前加一个反斜杠时会拥有不同的含义。
- 标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。
- 反斜杠本身需要使用反斜杠转义。
- 正则表达式通常都包含反斜杠，所以最好**使用原始字符串**来表示它们，如：`r'\t'`等价于`\t`。

正则表达式模式语法中的特殊元素表: (若使用模式的同时提供了可选的标志参数，则某些模式元素的含义会改变)

Pattern	Description
^	匹配字符串的开头
$	匹配字符串的末尾
.	匹配除了换行符的任意字符 (当`re.DOTALL`标记被指定时也可匹配换行符)
[...]	用来表示一组字符并单独列出，如: [amk] 匹配 'a', 'm'或'k'
[^...]	不在[]中的字符，如: [^abc] 匹配除了'a', 'b', 'c'之外的字符
*	匹配 0 个或多个的表达式
+	匹配 1 个或多个的表达式
?	匹配 0 个或 1 个由前面的正则表达式定义的片段，非贪婪方式
{n}	匹配 n 个前面表达式，如: "o{2}"不能匹配"Bob"中的"o"，但是能匹配"food"中的两个o
{n,}	精确匹配 n 个前面表达式，如: "o{2,}"不能匹配"Bob"中的"o"，但能匹配"fooooood"中的所有o。"o{1,}"等价于"o+"，"o{0,}"则等价于"o*"。
{n, m}	匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
a | b	匹配 a 或 b
(re)	匹配括号内的表达式，也表示一个组
(?imx)	正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。
(?-imx)	正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。
(?: re)	类似 (...), 但是不表示一个组
(?imx: re)	在括号中使用i, m, 或 x 可选标志
(?-imx: re)	在括号中不使用i, m, 或 x 可选标志
(?#...)	注释
(?= re)	前向肯定界定符。若所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
(?! re)	前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功。
(?> re)	匹配的独立模式，省去回溯
\w	匹配数字字母下划线
\W	匹配非数字字母下划线
\s	匹配任意空白字符，等价于 [\t\n\r\f]
\S	匹配任意非空字符

Pattern	Description
\d	匹配任意数字，等价于 [0-9]
\D	匹配任意非数字
\A	匹配字符串开始
\Z	匹配字符串结束，若存在换行，则只匹配到换行前的结束字符串
\z	匹配字符串结束
\G	匹配最后匹配完成的位置
\b	匹配一个单词和特殊字符边界，如: 'er\b' 可匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'
\B	匹配非单词边界。如: 'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'
\n, \t, \r	匹配一个换行符，匹配一个制表符，等等
\1...\9	匹配第 n 个分组的内容
\10	匹配第 n 个分组的内容，如果它经匹配。否则指的是八进制字符码的表达式

6.5 正则表达式实例

6.5.1 字符匹配

Example	Description
python	匹配 "python".

6.5.2 字符类

Example	Description
[Pp]ython	匹配 "Python" 或 "python"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配中括号内的任意一个字母
[0-9]	匹配任何数字。类似于 [0123456789]
[a-z]	匹配任何小写字母
[A-Z]	匹配任何大写字母
[a-zA-Z0-9]	匹配任何字母及数字
[^aeiou]	除了括号内的字母以外的所有字符
[^0-9]	匹配除了数字外的字符

6.5.3 特殊字符类

Example	Description
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用 '[.\n]' 的模式。
\d	匹配一个数字字符，等价于 [0-9]
\D	匹配一个非数字字符，等价于 [^0-9]
\s	匹配任何空白字符，包括空格、制表符、换页符等等，等价于 [\f\n\r\t\v]
\S	匹配任何非空白字符，等价于 [^ \f\n\r\t\v]
\w	匹配包括下划线的任何单词字符，等价于 [A-Za-z0-9_]
\W	匹配任何非单词字符，等价于 '[^A-Za-z0-9_]'
\\d	

6.5.4 其他说明

1. 使用转义符号' \ '可匹配模式本身，如: ' \\d ' 可匹配 ' \d '本身。
2. 使用`(?<name>exp)`可匹配`exp`并捕获文本到名称为`name`的组里。在 Python 中为`(?P<name>exp)`：

```
>>> import re
>>> pattern = re.compile(r'(?P<here>[a-z]+) ([a-z]+)', re.I)
>>> m = pattern.match('Hello World word helo')
>>> print (m.group('here'))
Hello
```

7 使用`json`模块解析数据

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式，是基于 ECMAScript 的一个子集。Python 3 中可使用`json`模块来对 JSON 数据进行编解码，它包含了两个函数：

- `json.dump(obj, fp, *, cls=None, indent=None, separators=None, sort_keys=False)`对数据进行编码。
- `json.loads(fp, *, cls=None, object_hook=None, object_pairs_hook=None,)`: 对数据进行解码。

7.1 Python 编码与 JSON 类型转换表

在`json`的编解码过程中，Python 的原始类型与`json`类型会相互转换，具体的转化对照如下：

Python	JSON
`dict`	`object`
`list`, `tuple`	`array`
`str`	`string`
`int`, `float`, & -derivated Enums	`number (int)`, `number (real)`
`True`	`true`
`False`	`false`
`None`	`null`

7.2 `json.dumps` 与 `json.loads` 实例

以下实例演示了 Python 数据结构转换为 JSON：

```
#!/usr/bin/env python3
import json
# Python 字典类型转换为 JSON 对象
data = {
    'no' : 1,
    'name' : 'Runoob',
    'url' : 'http://www.runoob.com'
}
json_str = json.dumps(data)
print ("Python 原始数据：", repr(data))
print ("JSON 对象：", json_str)
```

执行以上代码输出结果为：

```
Python 原始数据： {'url': 'http://www.runoob.com', 'no': 1, 'name': 'Runoob'}
JSON 对象： {"url": "http://www.runoob.com", "no": 1, "name": "Runoob"}
```

通过输出的结果可以看出，简单类型通过编码后跟其原始的`repr()`输出结果非常相似。

接着以上实例，我们可以将一个 JSON 编码的字符串转换回一个 Python 数据结构：

```
#!/usr/bin/env python3
import json
# Python 字典类型转换为 JSON 对象
data1 = {
    'no' : 1,
    'name' : 'Runoob',
    'url' : 'http://www.runoob.com'
}
json_str = json.dumps(data1)
print ("Python 原始数据：", repr(data1))
print ("JSON 对象：", json_str)
# 将 JSON 对象转换为 Python 字典
data2 = json.loads(json_str)
print ("data2['name']: ", data2['name'])
print ("data2['url']: ", data2['url'])
```

执行以上代码输出结果为：

```
Python 原始数据： {'name': 'Runoob', 'no': 1, 'url': 'http://www.runoob.com'}
JSON 对象： {"name": "Runoob", "no": 1, "url": "http://www.runoob.com"}
data2['name']:  Runoob
data2['url']:  http://www.runoob.com
```

如果你要处理的是文件而不是字符串，你可以使用 **json.dump()** 和 **json.load()** 来编码和解码 JSON 数据。如：

```
# 写入 JSON 数据
with open('data.json', 'w') as f:
    json.dump(data, f)

# 读取数据
with open('data.json', 'r') as f:
    data = json.load(f)
```

8 使用`unittest`测试模块