



Python3 快速入门

作者：李振良

关于本文档

文档名称	Python3 快速入门
联系作者	李振良（阿良），微信：k8init
官方网站	http://www.aliangedu.cn
DevOps 开发群	849855273
说明	本文档均为个人经验总结，转发请保留出处，拒绝不道德行为。 文档会不定期修改或新增知识点，请关注群动态或微信公众号获取最新文档。 如果文档有错误，欢迎联系阿良指正！
最后更新时间	2021-2-29



阿良个人微信



DevOps技术栈公众号

第一章	Python 基础知识.....	7
1.1	介绍.....	7
1.2	Linux 安装 Python.....	7
1.3	代码规范化.....	8
1.4	交互式解释器.....	8
1.5	运行第一个程序.....	9
1.6	基本数据类型.....	9
1.7	算数运算符.....	9
1.8	赋值操作符.....	10
1.9	变量.....	10
1.10	转义字符.....	12
1.11	获取用户输入.....	13
1.12	注释.....	13
第二章	Python 字符串.....	14
2.1	字符串格式化输出.....	14
2.1	字符串拼接.....	14
	join().....	14
2.1	计算字符串长度.....	15
2.1	字符串切片.....	15
2.1	字符串处理方法.....	16
2.1	字符串输出颜色.....	17
第三章	Python 数据类型.....	18
3.1	列表[List].....	18
3.1.1	定义列表.....	18
3.1.2	基本操作.....	18
3.1.4	切片.....	20
3.1.5	清空列表.....	20
3.2	元组(Tuple).....	20
3.3	集合(set).....	21
3.3.1	定义集合.....	21
3.3.2	基本操作.....	22
3.3.3	关系测试.....	22
3.4	字典{Dict}.....	23
3.4.1	定义字典.....	23
3.4.2	基本操作.....	23
3.4.4	字典嵌套.....	25
3.5	常见数据类型转换.....	25
第四章	Python 运算符和流程控制.....	27
4.1	基本运算符.....	27
4.1.1	比较操作符.....	27
4.1.2	逻辑运算符.....	27
4.1.3	成员运算符.....	28
4.1.4	身份运算符.....	29
4.2	条件判断.....	29
4.2.1	单分支.....	30
4.2.2	多分支.....	30
4.3	循环语句.....	31
4.3.1	for.....	31

4.3.2 range() 函数.....	33
4.3.2 while.....	34
4.3.3 continue 和 break 语句.....	35
4.3.4 循环中的 else 子句.....	37
4.2.3 pass 语句.....	37
第五章 Python 文件操作.....	38
5.1 open() 函数.....	38
5.2 文件对象操作.....	39
5.3 文件对象增删改查.....	41
5.4 with 语句.....	44
第六章 Python 函数.....	44
6.1 函数定义与调用.....	44
6.2 函数参数.....	45
6.3.1 接收参数.....	45
6.3.2 参数默认值.....	46
6.3.3 接受任意数量参数.....	46
6.3 匿名函数 (Lambda 表达式).....	47
6.4 作用域.....	47
6.4.1 全局和局部作用域.....	48
6.4.2 内置作用域.....	49
6.4.3 嵌套作用域.....	49
6.6 闭包.....	49
6.8 函数装饰器.....	50
6.8.1 无参数装饰器.....	50
6.8.2 带参数装饰器.....	51
第七章 Python 常用内建函数.....	52
7.1 高阶函数.....	52
7.1.1 map().....	53
7.1.2 filter().....	54
7.2 排序函数.....	54
7.2.1 sorted().....	54
7.2.3 reversed().....	55
7.3 最小最大值.....	55
7.3.1 min().....	55
7.3.2 max().....	56
7.4 求和.....	56
7.4 可迭代对象计数.....	56
7.5 多个迭代对象聚合.....	57
7.6 字符串转换表达式 eval().....	57
7.7 获取当前所有变量.....	57
第八章 Python 类 (面向对象编程).....	58
8.1 类的定义.....	58
8.2 类的书写规范.....	59
8.3 类实例化.....	60
8.4 属性操作.....	62
8.4.1 访问、修改、添加、删除.....	62
8.4.2 属性私有化.....	62
8.4.3 内置属性 (首尾双下划线).....	63
8.5 方法.....	64

8.6 类的继承	64
8.7 内置函数访问对象属性	66
8.7.1 getattr()	66
8.7.2 hasattr()	67
8.7.3 setattr()	67
8.7.4 delattr()	67
8.8 类装饰器	68
8.8.1 自定义装饰器	68
8.8.2 类内置装饰器	68
第九章 Python 异常处理	70
9.1 捕捉异常语法	70
9.2 异常类型	71
9.3 异常处理	73
9.4 else 和 finally 语句	74
9.4.1 else 语句	74
9.4.2 finally 语句	74
9.4.3 try...except...else...finally	75
9.5 raise 自定义异常	75
第十章 Python 自定义模块及导入方法	76
10.1 自定义模块	76
10.2 <code>__name__ == '__main__'</code> 作用	78
10.3 模块帮助文档	78
10.4 导入模块新手容易出现的问题	79
第十一章 Python 常用标准库使用	79
11.1 os (操作系统管理)	80
11.2 sys (解释器交互)	82
11.3 platform (获取操作系统信息)	86
11.4 glob (查找文件)	87
11.5 fileinput (处理多文件)	87
11.6 shutil (文件管理)	88
11.7 math (数字)	89
11.8 random (随机数)	90
11.9 subprocess (执行 Shell 命令)	90
11.10 pickle (数据持久化)	91
11.11 json (JSON 编码和解码)	92
11.12 time (时间访问和转换)	93
11.13 datetime (日期和时间)	95
11.14 urllib (HTTP 访问)	98
11.15 configparser (文件格式)	104
11.16 argparse (命令行参数解析)	107
11.17 smtplib (发送邮件)	109
第十二章 Python 正则表达式	118
12.1 re 模块的基本使用	118
12.2 代表字符	119
12.3 原始字符串符号“r”	121
12.4 代表数量	122
12.5 代表边界	124
12.6 代表分组	125
12.7 贪婪和非贪婪匹配	127

12.8 其他方法	128
search()	128
findall()	128
split()	128
sub()	128
12.9 标志位	129
第十三章 Python 数据库编程	130
13.1 PyMySQL	130
13.1.1 安装 pymysql 模块	130
13.1.2 pymysql 使用	130
13.1.3 数据库增删改查	132
13.1.4 遍历查询结果	134
13.2 SQLAlchemy	135
13.2.1 安装 sqlalchemy 模块	135
13.2.2 基本使用	135
第十四章 Python 批量管理主机	138
14.1 paramiko	138
14.1.1 SSH 密码认证远程执行命令	139
14.1.2 SSH 密钥认证远程执行命令	139
14.1.3 上传文件到远程服务器	140
14.1.4 上传目录到远程服务器	141
14.1.5 从远程服务器下载文件	143
14.2 pexpect	144

第一章 Python 基础知识

1.1 介绍

Python 特点

Python 是一种面向对象、解释型、多用途设计语言，具有很丰富和强大的库，语法简洁，强制用空格作为语法缩进，能够完成快速项目开发，相比传统语言开发效率提高数倍。也称为胶水语言，能把其他语言（主要 C/C++）写的模块很轻松的结合在一起。

Python 可以做什么？

- **Web 网站：**有很多优秀的开源 Web 框架，比如 Django（最流行，企业级框架）、Tornado（轻量级、异步）、Flask（微型）、Web.py（简单）等。
- **数据采集：**对网站数据抓取有一套成熟的库，比如 HTTP 客户端库 urllib2、requests 等；网页解析库 lxml、xpath、BeautifulSoup 等；还有高级的屏幕爬取及网页采集框架 scrapy。
- **大数据分析：**常用模块有 Numpy、Pandas。支持编写 MapReduce 任务、PySpark 处理 Spark RDD（弹性分布式数据集）的接口库。
- **系统运维：**内置许多系统标准库，可以很方面编写运维常规任务脚本。
- **科学计算：**在科学计算也应用越来越广泛，常用的模块有 Numpy、SciPy。
- **用户图形：**开发桌面 GUI 程序，内置 TKinter 标准接口、wxPython。

等等...可见 Python 是一门通用语言，在多个领域都得到了广泛使用！

为什么选择 Python？

- 语法简洁，易于学习
- 广泛的标准库，适合快速开发
- 跨平台，基本所有的操作系统都能运行
- Python 在运维领域最流行

对于运维来说，Python 是你首选的开发语言，因为它易学易用，能够满足大部分自动化需求，快速开发出高大上的运维管理平台，是目前系统运维应用最广泛的语言，没有之一！

1.2 Linux 安装 Python

安装依赖包：

```
CentOS: yum install zlib-devel -y
```

```
Ubuntu: apt-get install zlib1g-dev -y
```

下载安装包并编译安装：

```
# wget https://www.python.org/ftp/python/3.6.0/Python-3.6.0.tar.xz
# tar xvf Python-3.6.0.tar.xz
# cd Python-3.6.0
# ./configure --enable-optimizations
# make && make install
```

查看 python 版本：

```
import platform
print(platform.python_version())
或者
python -V
```

1.3 代码规范化

代码规范重要性

- 团队协作

在企业中，往往是团队开发一个项目，按照模块、前后端分工。项目立项后，开发组制定接口、文档等方面标准，其目的让参与项目中的每位成员，在写代码时能够统一标准，提高可控性，避免项目中出现多种编码风格版本不利于对接及后期维护。

- 有利于解决问题

项目开发完成，发布到线上运行，谁也保证不了不出问题，自己写的代码出问题搜索引擎也帮不了你！逻辑复杂，注释不明确，不得不从头梳理代码和逻辑关系，增加处理时间和排查难度。最后可能是一段代码编写不严谨导致，真是浪费了大把时间！

编写代码怎么能更规范化？

1) 缩进

Python 以空白符作为语句缩进，意味着语句没有结尾符，刚入门的朋友往往因为上下逻辑代码不对齐导致运行报错，在 Python 中最好以 4 个空格作为缩进符，严格对齐。

2) 注释

据说优质的代码，注释说明要比代码量多，详细的代码说明不管是对自己还是对他人，在后期维护中都是非常有利的。就像一个流行的开源软件，如果没有丰富的使用文档，你认为会有多少人去花大把时间研究它呢！

3) 空格

在操作符两边，以及逗号后面，加 1 个空格。但是在括号左右不加空格。

在函数、类、以及某些功能代码块，空出一行，来分隔它们。

4) 命名

模块：自己写的模块，文件名全部小写，长名字单词以下划线分隔。

类：大/小驼峰命名。我一般采用大驼峰命名，也就是每个单词首字母大写。类中私有属性、私有方法，以双下划线作为前缀。

函数：首单词小写，其余首字母大写。

变量：都小写，单词以下划线分隔。关键字不能用变量吗

所有的命名规则必须能简要说明此代码意义。

5) 换行

按照语法规则去换行，比如一个很长的表达式，可以在其中某个小表达式两边进行换行，而不是将小表达式拆分，这样更容易阅读。

1.4 交互式解释器

在 Linux 终端执行 Python 命令就启动默认的 CPython 解释器：

```
# python3.6
Python 3.6.0 (default, Apr 10 2017, 01:28:54)
[GCC 6.2.0 20161005] on linux
Type "help", "copyright", "credits" or "license" for more information.
```



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

1.5 运行第一个程序

打开一个文件，以.py 为后缀。

```
# vim hello.py
#!/usr/bin/python
print("Hello World!")

# python3.7 hello.py
Hello World!
```

文件开头一般需要指定 python 执行程序，也可以使用 env 会自动搜索变量找到 python 解释器。

1.6 基本数据类型

什么是数据类型？

我们人类可以很容易的分清数字与字符的区别，但是计算机并不能呀，计算机虽然很强大，但从某种角度上看又很傻，除非你明确的告诉它，1 是数字，“汉”是文字，否则它是分不清 1 和‘汉’的区别的，因此，在每个编程语言里都会有一个叫数据类型的东东，其实就是对常用的各种数据类型进行了明确的划分，你想让计算机进行数值运算，你就传数字给它，你想让他处理文字，就传字符串类型给他。

Python 中常用的基本数据类型有：

- 整数（int），例如 6
- 浮点数（float），例如 6.6
- 字符串（str），例如"8","python"
- 布尔值（bool），例如 True、False

可以使用 type() 内置函数查看数据类型。

1.7 算数运算符

什么是运算符？

举个简单的例子 $6+6=12$ ，其中两个 6 被称为操作符，+称为运算符。

运算符	描述	示例
+	加法	(6 + 6) 结果 12
-	减法	(6 - 6) 结果 0
*	乘法	(6 * 6) 结果 36
/	除	(8 / 6) 结果 1.33333333

//	整除	(8 / 6)结果 1
%	取余	(6 % 6)结果 0
**	幂	(6 ** 3)结果 46656，即 6 * 6 * 6
()	小括号	小括号用来提高运算优先级，即 (1+2)*3 结果为 9

1.8 赋值操作符

运算符	描述	示例
=	赋值，将=左侧的结果赋值给等号左侧的变量	a = 10 b = 20
+=	加法赋值	c += a 等价 c = c + a
-=	减法赋值	c -= a 等价 c = c - a
*=	乘法赋值	c *= a 等价 c = c * a
/=	除法赋值	c /= a 等价 c = c / a
//=	整除赋值	c //= a 等价 c = c // a
%=	取余赋值	c %= a 等价 c = c % a
**=	幂赋值	c **= a 等价 c = c ** a

先计算，再赋值。

1.9 变量

1.8.1 变量赋值

```
name = "aliang"  
print(name)
```

这里定义了一个变量名为 name，值为 aliang，使用 print 函数打印变量值。
也可以同时给多个变量赋值（多重赋值）：

```
name1, name2 = "aliang", "lizhenliang"  
print(name1)  
print(name2)  
# 运行结果  
aliang  
lizhenliang
```

多个变量同时赋予一个值：

```
name1 = name2 = "aliang"
print(name1)
# 运行结果
aliang
```

1.8.2 变量引用

上面打印就是引用的变量，可见 Python 引用变量直接使用变量名。不像 Shell、PHP 那样，要加\$。再看看另一种常用的引用方法，在输出字符串时引用变量：

```
name = "aliang"
print("姓名： %s" %name)
# 运行结果
姓名： aliang
```

同时引用多个变量：

```
name = "aliang"
age = 30
print("姓名： %s, 年龄： %d" %(name, age))
# 运行结果
姓名： aliang, 年龄： 30
```

%是一个占位符，外面%()里为变量名，位置逐一对应双引号里的%位置。

- %s 表示字符串 str，
- %d 表示整数 int，还可以表示浮点数 float
- %f 表示浮点数 float，还可以表示整数 int

操作符号	描述
%s	字符串（str()）
%r	字符串（repr()）
%d	整数
%f	浮点数，可指定小数点后的精度

1) 字符串格式输出三种方法

```
>>> xxoo = "string"
>>> print "%s" %xxoo
string
>>> print "%r" %xxoo
'string'
>>> print `xxoo`
'string'
```

%s 采用 str() 函数显示，%r 采用 repr() 函数显示。repr() 和反撇号把字符串转为 Python 表达式。

2) 保留小数点数

```
>>> '%.1f' %(float(100)/1024)
```

```
'0.1'
```

1.10 转义字符

当你输入一些特定的字符时，Python 会按照预定的含义进行解释并输出，例如字符串中包含双引号：

```
print("姓名： %s, "年龄： %d" %(name, age))
```

运行结果

```
SyntaxError: invalid character in identifier
```

提示语法错误，这是因为 Python 不知道以那个双引号结束，此时则需要对多的双引号进行转义：

```
print("姓名： %s, \"年龄： %d\" %(name, age))
```

其中\称为转义符，这里将双引号失去本身的意义，作为普通字符串使用。
以下是常用转义符。

转义字符	说明
\n	换行符，将光标位置移到下一行开头。
\r	回车符，将光标位置移到本行开头。
\t	水平制表符，也即 Tab 键，一般相当于四个空格。
\b	退格（Backspace），将光标位置移到前一列。
\\	反斜线
\'	单引号
\"	双引号
\\	在字符串行尾的续行符，即一行未完，转到下一行继续写。

示例：

```
print("Hello \nWorld!")
print("Hello \tWorld!")
print("Hello \\World!")
print("Hello \
World!")
# 运行结果
Hello
World!
Hello  World!
Hello \World!
Hello World!
```

如果不想让转义字符生效，可以用 r 指定显示原始字符串：

```
print(r"Hello \nWorld!")  
# 运行结果  
Hello \nWorld!
```

print 结束符是\n 换行符，也可以定义别的符号：

```
print("hello", end="\t")  
print("hello", end="\n")  
# 运行结果  
hello  hello
```

1.11 获取用户输入

有时候，编写的程序需要从用户那儿“拿到”一些数据才能继续执行下去，比如，判断某人是否到了法定结婚年龄，需要用户自己输入年龄才可以。

```
age = input("小盆友，今年多大啦？\n 请输入你的年龄：")  
print("呦，都%s 岁了！可以谈恋爱了。" %age)  
# 运行结果  
小盆友，今年多大啦？  
请输入你的年龄：18  
呦，都 18 了！可以谈恋爱了。
```

函数 `input()` 用于与用户交互，接收一个参数，即要向用户显示的提示或者说明，让用户知道该怎么做。

在上述示例中 `age` 为变量名，变量值为用户输入的值，通过变量 `age` 拿到用户输入的值，然后根据这个值就可以继续完成对应操作了。

1.12 注释

单行注释：井号（“#”）开头

多行注释：三单引号或三双引号

```
#!/usr/bin/env python
```

```
# 单行注释
```

```
'''
```

```
多行注释
```

```
多行注释
```

```
'''
```

```
"""
```

```
多行注释
```

```
多行注释
```

```
"""
```

第二章 Python 字符串

字符串是 Python 中最常用的数据类型，字符串创建由单引号或者双引号括起来，正如前面定义的名称="aliang" 这样。

本章节主要对字符串进行处理。

2.1 字符串格式化输出

```
name = "aliang"
age = 30
# 方法 1
print("我的名字是%s, 今年%s 岁了。"% (name, age))
# 方法 2
print(f"我的名字是{name}, 今年{age} 岁了。")
```

注：f' {表达式}' 格式化字符串是 Python3.6 中新增的格式化方法，该方法更简单易读。

2.1 字符串拼接

使用 "+" 可以对多个字符串进行拼接。

```
str1 = "lizhen"
str2 = "liang"
print(str1 + str2)
# 运行结果
lizhenliang
```

需注意，字符串不允许直接与其他数据类型进行拼接，例如

```
str1 = "hello"
num = 3
print(str1 + num)
# 运行结果
TypeError: can only concatenate str (not "int") to str
```

上面这种情况我们可以将 num 值通过 str() 函数转换为字符串再进行拼接：

```
print("hello" + str(num))
```

或者以变量方式引用：

```
res = "%s%d" %(str1, num)
print(res)
```

join()

join() 方法用于将序列中年的元素以指定的字符连接生成一个新的字符串。

语法格式：str.join(iterable) # 其中 str 是要分隔的字符，iterable 是可迭代对象

```
s = "lizhenliang"
r = '.'.join(s)
print(r)
# 运行结果
l.i.z.h.e.n.l.i.a.n.g

computer = ["主机", "显示器", "鼠标", "键盘"]
r = ','.join(computer)
print(r)
# 运行结果
主机, 显示器, 鼠标, 键盘
```

字符串拼接尽量多用 join，相比+性能更好

2.1 计算字符串长度

使用 len() 函数计算字符串的长度。

语法格式：len(string)

```
str1 = "lizhenliang"
print(len(str1))
# 运行结果
11
```

2.1 字符串切片

语法格式： string[start : end : step]

- string 表示要切片的字符串
- start 表示要切片的第一个字符串的索引（包括该字符），如果不指定默认为 0
- end 表示要切片的最后一个字符的索引（不包括该字符），如果不指定默认为字符串的长度
- step 表示切片的步长，如果不指定默认为 1

```
str1 = "Hello World!"

# 从 1 个字符到第 5 个字符截取
s = str1[0:5]
print(s)
# 截取第 5 个字符
s = str1[4]
print(s)
# 截取最后一个字符
s = str1[-1]
print(s)

# 从第 6 个字符到倒数第 2 个字符
s = str1[6:-2]
```

```
# 运行结果
Hello
o
!
Worl
```

2.1 字符串处理方法

下面是一些常用的字符串处理方法，红色颜色标记的更为常用。
举例说明：

```
#!/usr/bin/python

xxoo = "abcdef"
print("首字母大写: %s" % xxoo.capitalize())
print("字符 l 出现次数: %s" % xxoo.count('l'))
print("感叹号是否结尾: %s" % xxoo.endswith('!'))
print("w 字符是否是开头: %s" % xxoo.startswith('w'))
print("w 字符索引位置: %s" % xxoo.find('w')) # xxoo.index('W')
print("格式化字符串: Hello{0} world!".format(','))
print("是否都是小写: %s" % xxoo.islower())
print("是否都是大写: %s" % xxoo.isupper())
print("所有字母转为小写: %s" % xxoo.lower())
print("所有字母转为大写: %s" % xxoo.upper())
print("感叹号替换为句号: %s" % xxoo.replace('!', '.'))
print("以空格分隔切分成列表: %s" % xxoo.split(' '))
print("切分为一个列表: %s" % xxoo.splitlines())
print("去除两边空格: %s" % xxoo.strip())
print("大小写互换: %s" % xxoo.swapcase())
```

```
# 运行结果
字符串长度: 6
首字母大写: Abcdef
字符 l 出现次数: 0
感叹号是否结尾: False
w 字符是否是开头: False
w 字符索引位置: -1
格式化字符串: Hello, world!
是否都是小写: True
是否都是大写: False
所有字母转为小写: abcdef
所有字母转为大写: ABCDEF
感叹号替换为句号: abcdef
以空格分隔切分成列表: ['abcdef']
切分为一个列表: ['abcdef']
去除两边空格: abcdef
```


2.1 字符串输出颜色

字体颜色	字体背景颜色	显示方式
30: 黑 31: 红 32: 绿 33: 黄 34: 蓝色 35: 紫色 36: 深绿 37: 白色	40: 黑 41: 深红 42: 绿 43: 黄色 44: 蓝色 45: 紫色 46: 深绿 47: 白色	0: 终端默认设置 1: 高亮显示 4: 下划线 5: 闪烁 7: 反白显示 8: 隐藏
<p>格式: \033[1;31;40m # 1 是显示方式，可选。31 是字体颜色。40m 是字体背景颜色。 \033[0m # 恢复终端默认颜色，即取消颜色设置。</p>		

示例:

```
#!/usr/bin/python
# 字体颜色
for i in range(31, 38):
    print("\033[%s;40mHello world!\033[0m" %i)
# 背景颜色
for i in range(41, 48):
    print("\033[47;%smHello world!\033[0m" %i)
# 显示方式
for i in range(1, 9):
    print("\033[%s;31;40mHello world!\033[0m" %i)
```

什么是数组？

在 Python 中，内建数据结构有列表（list）、元组（tuple）、字典（dict）、集合（set）。

3.1.1 定义列表

```
computer = ["主机", "显示器", "鼠标", "键盘"]
```

用中括号括起来，里面每个值为一个元素，以逗号分隔，中文、字符串用引号引起来，整数不用。

3.1.2 基本操作

```
# 追加一个元素
computer = ["主机", "显示器", "鼠标", "键盘"]
print(computer)
computer.append("音响")
print(computer)
```

```
# 运行结果
['主机', '显示器', '鼠标', '键盘']
['主机', '显示器', '鼠标', '键盘', '音响']
# 在第 0 个索引位置插入一个元素
computer = ["主机", "显示器", "鼠标", "键盘"]
computer.insert(0, "电脑")
print(computer)
# 运行结果
['电脑', '主机', '显示器', '鼠标', '键盘']
```

删：

```
# 通过具体值删除
computer.remove("音响")
# 或者通过索引删除
computer.pop(4)
computer.pop() # 如果未指定索引值默认为最后一个元素
```

改：

```
# 修改索引 3 的值
computer[3] = "音响"
print(computer)
# 运行结果
['主机', '显示器', '鼠标', '音响']
```

查：

```
# 查看元素“显示器”的索引位置
computer = ["主机", "显示器", "鼠标", "键盘"]
i = computer.index("显示器")
print(i)
# 运行结果
1

# 统计“主机”元素在列表出现的次数
computer = ["主机", "显示器", "鼠标", "键盘"]
c = computer.count("主机")
print(c)
# 运行结果
1

# 倒序排列元素
computer.reverse()
print(computer)
# ['键盘', '鼠标', '显示器', '主机']

# 正向排序元素，此时 list 本身会被修改
computer.sort()
```

```
# 列表拼接
computer2 = ["音响"]
print(computer + computer2)
```

3.1.4 切片

```
# 返回第 1 个元素
computer[0]

# 返回第 1 个元素到第 3 个元素
computer[0:3]

# 返回最后一个元素
computer[-1]

# 返回从第 1 个元素到倒数第二个元素
computer[0:-1]

# 步长切片
computer[0:4:2]
# 运行结果
['主机', '鼠标']
```

3.1.5 清空列表

```
# 方法 1, 重新创建, 即重新初始化列表为空
computer = []

# 方法 2, del 语句清空列表元素, 如果不给出范围, 则默认删除所有元素, 例如 del computer[0:2] 删除第 1 个元素到第 2 个元素
del computer[:]

# 删除列表
del computer
```

一般量的列表可直接用 `computer = []` 快速清空释放内存。

3.2 元组(Tuple)

元组与列表类似, 不同之处在于元组中的元素不能修改。元组使用小括号, 列表使用方括号。
定义元素:

```
computer = ("主机", "显示器", "鼠标", "键盘")
```

此时你无法再原地改变此 tuple 对象。

元组的特点：

- 不支持添加元素【增】
- 不支持删除元素【删】
- 不支持修改元素【改】
- 支持 2 种方法查找元素【查】

元组与字符串类似，下标索引从 0 开始，可以进行截取，组合等。

```
# 根据下标查找元素
computer[0]
# 根据元素获取下标
computer.index("鼠标")
```

如果想删除或者修改元素该怎么办呢？
可以转成列表再修改，然后再转回元组。

```
computer = ("主机", "显示器", "鼠标", "键盘")
l = list(computer)
l.pop(3)
computer2 = tuple(l)
print(computer2)
# 运行结果
('主机', '显示器', '鼠标')
```

count() 和 index() 方法和切片使用方法与列表使用一样，这里不再赘述。

也支持不同元组之前拼接、del 语句删除整个元素。

3.3 集合(set)

集合是一个无序、不重复元素的序列，主要用于元素去重和关系测试。

集合对象还支持联合 (union)，交集 (intersection)，差集 (difference) 和对称差集 (symmetric difference) 数学运算。

set 作为一个无序的集合，不记录元素位置，因此不支持索引、切片。

3.3.1 定义集合

使用 set() 函数或者大括号 {} 创建集合。

```
computer = set()
或者
computer = {"主机", "显示器", "鼠标", "键盘"}
```

注意：set() 函数创建集合只能传一个参数，即一个元素，否则会报错。当时使用 {} 创建时必须指定元素，否则会被认为创建字典。

当有重复元素时会自动去重：

```
computer = set(["主机", "显示器", "鼠标", "键盘", "主机"])
print(computer)
# 运行结果
{'显示器', '主机', '鼠标', '键盘'}
```

3.3.2 基本操作

增：

```
computer = set()
computer.add("主机")
computer.add("显示器")
computer.add("鼠标")
print(computer)
# 运行结果
{'主机', '显示器', '鼠标'}
```

可以看到，添加的元素是无序的，并且不重复的。

删：

```
computer.remove("鼠标")
remove 当删除的元素不存在时会报错造成程序退出，这时可以使用 discard 优雅处理，即使不存在元素也不会异常。
computer.discard("鼠标 2")
# 删除第 1 个元素
computer.pop()
```

改和查不支持，一般会 for 语句遍历处理。

典型的应用场景：列表去重

```
computer = ["主机", "显示器", "鼠标", "键盘", "显示器", "鼠标"]
s = set(computer)
print(s)
# 运行结果
{'显示器', '鼠标', '键盘', '主机'}
```

3.3.3 关系测试

符号	描述
-	差集
&	交集
	合集、并集

!=	不等于
==	等于

示例：

```
a = set([1, 2, 3, 4, 5, 6])
b = set([4, 5, 6, 7, 8, 9])

# 返回 a 集合中元素在 b 集合没有的
print(a - b)
# 返回 b 集合中元素在 a 集合中没有的
print(b - a)
# 返回交集，即两个集合中一样的元素
print(a & b)
# 返回合集，即合并去重
print(a | b)
# 判断是否不相等
print(a != b)
# 判断是否相等

# 运行结果
{1, 2, 3}
{8, 9, 7}
{4, 5, 6}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
True
```

3.4 字典{Dict}

字典是一个常用的数据类型，用于存储具有映射关系的数据，例如电脑配件价格，主机：5000，显示器：1000，键盘 150，鼠标 60，这组数据看上去像两个列表，但这两个列表的元素之前有一定的关联关系，如果单纯使用两个列表来保存这组数据，则无法记录它们之间的关联关系。为保存这类数据，Python 提供字典存储，元素由字典由键和值组成，逗号分隔（Key:Value）。

访问也与字符串、列表、元组这些序列是以连续的整数位索引，而字典通过 key 来访问 value，因此字典中的 key 不允许重复。

3.4.1 定义字典

```
d = {'key1':value1, 'key2':value2, 'key3':value3}
```

用大括号 {} 括起来，一个键对应一个值，冒号分隔，多个键值逗号分隔。

3.4.2 基本操作

```
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
```

增：

```
# 添加键值
computer["音响"] = 90
print(computer)

# 添加新字典
c = {"音响": 90}
computer.update(c)
print(computer)
```

删：

```
# 删除指定键值
computer.pop("音响")
print(computer)

# 删除最后一对键值并返回
r = computer.popitem()
print(computer)
print(r)

# 运行结果
{'主机': 5000, '显示器': 1000, '鼠标': 60}
('键盘', 150)
```

改：

```
computer["键盘"] = 300
```

修改与添加新内容方法一样，当没有该键时新增，有了就覆盖。

查：

```
# 获取键的值
print(computer["主机"])

# 获取所有键
print(computer.keys())

# 获取所有值
print(computer.values())

# 获取所有键值
print(computer.items())

# 以上运行结果
5000
dict_keys(['主机', '显示器', '鼠标', '键盘'])
dict_values([5000, 1000, 60, 150])
dict_items([('主机', 5000), ('显示器', 1000), ('鼠标', 60), ('键盘', 150)])

# 获取键的值，如果键不存在就会抛出 KeyError 错误
computer["麦克风"]
```


运行结果

KeyError: '麦克风'

我们并不希望就此终止程序，这时可以设置 get 方法默认值，如果这个键存在就返回对应值，否则返回自定义的值

```
print(computer.get("麦克风", None))
```

判断键是否存在

```
"键盘" in computer
```

#如果键不存在，将会添加键并将值设为默认值

```
r = computer.setdefault("显示器")
```

```
print(computer)
```

```
print(r)
```

3.4.4 字典嵌套

值不但是可以写整数、字符串，也可以是其他数据类型，例如列表、元组、集合、字典，这样可满足一个键还包含其他属性，例如“主机”由多个硬件组成。

```
computer = {"主机": {"CPU": 1300, "内存": 400, "硬盘": 200}, "显示器": 1000, "鼠标": 60, "键盘": ["机械键盘", "薄膜键盘"]}
```

获取“主机”值

```
print(computer["主机"])
```

运行结果

```
{'CPU': 1300, '内存': 400, '硬盘': 200}
```

向“主机”字典添加值

```
computer["主机"]["显卡"] = 1500
```

```
print(computer)
```

运行结果

```
{'主机': {'CPU': 1300, '内存': 400, '硬盘': 200, '显卡': 1500}, '显示器': 1000, '鼠标': 60, '键盘': ['机械键盘', '薄膜键盘']}
```

向“键盘”列表添加值

```
computer["键盘"].append("其他")
```

```
print(computer)
```

运行结果

```
{'主机': {'CPU': 1300, '内存': 400, '硬盘': 200}, '显示器': 1000, '鼠标': 60, '键盘': ['机械键盘', '薄膜键盘', '其他']}
```

3.5 常见数据类型转换

函数	描述
<code>int(x[, base])</code>	将 x 转换为一个整数
<code>float(x)</code>	将 x 转换为一个浮点数

<code>str(x)</code>	将对象 x 转换为字符串
<code>tuple(seq)</code>	将序列 seq 转换为一个元组
<code>list(seq)</code>	将序列 seq 转换为一个列表

示例：

```
# 转整数
>>> i = '1'
>>> type(i)
<type 'str'>
>>> type(int(i))
<type 'int'>
# 转浮点数
>>> f = 1
>>> type(f)
<type 'int'>
>>> type(float(f))
<type 'float'>
# 转字符串
>>> i = 1
>>> type(i)
<type 'int'>
>>> type(int(1))
<type 'int'>
# 字符串转列表
方式 1:
>>> s = 'abc'
>>> lst = list(s)
>>> lst
['a', 'b', 'c']
方式 2:
>>> s = 'abc 123'
>>> s.split()
['abc', '123']
# 列表转字符串
>>> s = ""
>>> s = ''.join(lst)
>>> s
'abc'
# 元组转列表
>>> lst
['a', 'b', 'c']
>>> t = tuple(lst)
>>> t
('a', 'b', 'c')
# 列表转元组
>>> lst = list(t)
>>> lst
['a', 'b', 'c']
```

```
# 字典格式字符串转字典
>>> s = '{"a": 1, "b": 2, "c": 3}'
>>> type(s)
<type 'str'>
>>> d = eval(s)
>>> d
{'a': 1, 'c': 3, 'b': 2}
>>> type(d)
<type 'dict'>
```

第四章 Python 运算符和流程控制

在第一章的时候讲解了运算操作符和赋值操作符，这章来学习下其他常用操作符。

4.1 基本运算符

4.1.1 比较操作符

运算符	描述	示例
==	相等，两边值是否相等	(6 == 6) 结果 True
!=	不相等，两边值是否不相等	(6 != 6) 结果 False
>	大于，左边值是否大于右边值	(8 > 6) 结果 True
<	小于，左边值是否小于右边值	(8 < 6) 结果 False
>=	大于等于，左边值是否大于等于右边值	(6 >= 6) 结果 True
<=	小于等于，左边值是否小于等于右边值	(6 <= 6) 结果 True

4.1.2 逻辑运算符

逻辑运算符常用于表达式判断。

运算符	逻辑表达式	描述
and	x and y	与，所有的条件都 True 结果才为 True；只要有一个为 False，结果就为 False

or	x or y	或，所有的条件只要有一个是 True 结果就为 True
not	not x	非，结果取反

示例：

```
a = 1
b = 2
c = 3

# and, 都为真才为真
print(a < b and c > b )
print(a < b and c < b)
# 运行结果
True
False

# or, 一个为真则为真，都为假才为假
print(a < b or c > b)
print(a > b or c > b)
# 运行结果
True
True

# not, 取反
print(not False)
print(not c > b)
# 运行结果
True
True
```

4.1.3 成员运算符

成员运算符用于测试数据类型（字符串、列表、元组、字典等）中是否包含值。

运算符	描述	示例
in	如果在指定的序列中找到值返回 True，否则返回 False	computer = ["主机", "显示器", "鼠标", "键盘"] ("主机" in computer) 结果 True ("音响" in computer) 结果 False
not in	如果在指定的序列中没有找到值返回 True，否则返回 False	print("主机" not in computer) 结果 False print("音响" not in computer) 结果 True

4.1.4 身份运算符

运算符	描述
is	判断两个标识符是否引用一个对象
is not	判断两个标识符是否引用不同对象

示例：

```
>>> a = []
>>> b = []
>>> id(a)
139741563903296
>>> id(b)
139741563902144
>>> a is b
False
>>> a is not b
True
```

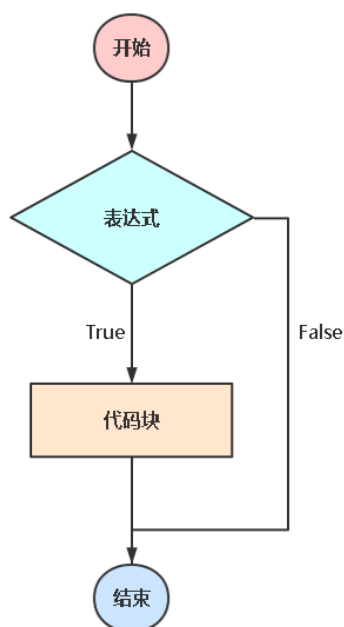
这里用到了 `id()` 函数，用于获取对象在内存的地址。

is 与 “==” 区别：

`is` 是比较对象是否相同(`is` 表示对象标识符即 object identity)，即用 `id()` 函数查看的地址是否相同，如果相同则返回 `True`，如果不同则返回 `False`。`is` 不能被重载。

“`==`” 是比较两个对象的值是否相等，此操作符内部调用的是 “`__eq__()`” 方法。所以 `a==b` 等效于 `a.__eq__(b)`，所以 “`==`” 可以被重载。

4.2 条件判断



编程语言中最为人所熟知的语句就是 if 了。

语法格式：

```
if 表达式:  
    代码块  
elif 表达式:  
    代码块  
else:  
    代码块
```

4.2.1 单分支

判断你是否成年。

```
age = int(input("请输入你的年龄："))  
if age > 18:  
    print("恭喜，你已经成年！")  
else:  
    print("抱歉，你还未成年！")
```

else 部分可以选，也可以不写。

条件判断也有简写的语法，称为三目表达式。

```
result = "恭喜，你已经成年！" if age > 18 else "抱歉，你还未成年！"  
print(result)  
# 运行结果  
抱歉，你还未成年！
```

结果生成的是一个字符串。

```
print(type(result))  
<class 'str'>
```

也可以使用[]结果生成的是一个列表。

```
result = ["恭喜，你已经成年！" if age > 18 else "抱歉，你还未成年！"]  
print(result)  
print(type(result))  
# 运行结果  
['抱歉，你还未成年！']  
<class 'list'>
```

4.2.2 多分支

根据人的年龄段划分儿童、少年、青年、壮年、老年。

```
age = int(input("请输入你的年龄："))  
  
if age < 7 :  
    print("儿童")
```

```
elif age >= 7 and age < 17:
    print("少年")
elif age >= 18 and age < 40:
    print("青年")
elif age >= 41 and age < 48:
    print("壮年")
else:
    print("老年")
```

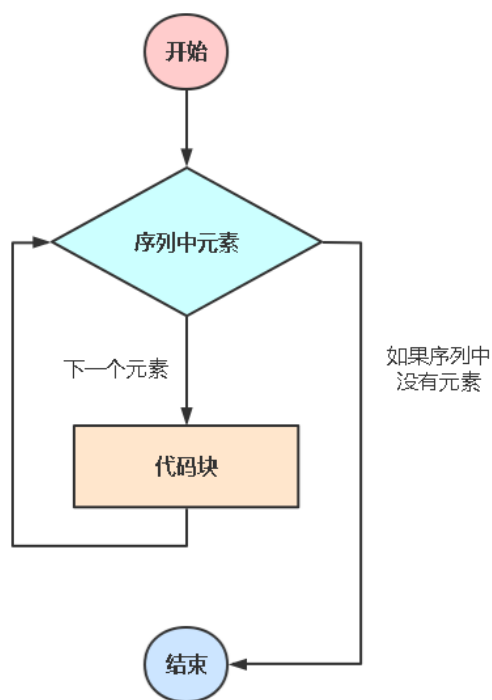
可以有零个或多个 elif 部分，关键字 elif 是 else if 的缩写，避免过多的缩进。

4.3 循环语句

4.3.1 for

for 语句可以对任意序列进行迭代（例如字符串、列表、字典等），条目的迭代顺序与它们在序列中出现的顺序一致。不能对数值进行迭代。

语法格式：



遍历字符串，每个字符代表一个元素：

```
s = "123456"
for i in s:
    print(i)
# 运行结果
1
2
3
4
5
```

6

嵌套循环：打印两个字符串中相同的字符

```
s1 = "123456"
s2 = "456789"
for i in s1:
    for x in s2:
        if i == x:
            print(i)
# 运行结果
4
5
6
```

for 语句也有简写的语法，例如将字符串生成一个列表。

```
s = "123456"
result = [x for x in s]
print(result)
# 运行结果
['1', '2', '3', '4', '5', '6']
```

for 和 if 语句写一行：

```
result = [x for x in s if int(x) % 2 == 0]
print(result)
# 运行结果
['2', '4', '6']
```

for 每一次迭代，都会把迭代对象放到 x 变量中，此时 x 变量值为字符串需要转型为整数，如果该值能与 2 整除，就保留该 x 值，最后生成一个列表。

示例 1：遍历列表，打印元素和元素长度

```
computer = ["主机", "显示器", "鼠标", "键盘"]
for i in computer:
    print(i, len(i))
# 运行结果
主机 2
显示器 3
鼠标 2
键盘 2
```

示例 2：遍历字典，分别打印键和值

```
computer = {"主机":5000, "显示器":1000, "鼠标":60, "键盘":150}
for i in computer.items():
```



```
print(i)
print("名称: %s\t 价格: %s" % (i[0], i[1]))
```

`print(i)` 结果返回是元组，例如（'主机'，5000）
`i[0]` 对元组切片获取第一个元素（键），`i[1]` 获取第二个元素（值）
也可以多重赋值，例如 `k, v = ('主机', 5000)`

```
for k, v in computer.items():
    print("名称: %s\t 价格: %s" % (k, v))
```

返回结果与上面一样。

4.3.2 range() 函数

正如上面变量 `s = "123456"` 字符串序列，如果你需要遍历一个数字序列，内置函数会派上用场。

```
for i in range(5):
    print(i)
# 运行结果
0
1
2
3
4
```

生成的数值从 0 开始，也可以指定开始值、或者指定步长。

```
range(5, 10)
5, 6, 7, 8, 9

range(0, 10, 3)
0, 3, 6, 9
```

也可以使用序列的索引来迭代。

```
computer = ["主机", "显示器", "鼠标", "键盘"]
for i in range(len(computer)):
    print(i, computer[i])
# 运行结果
0 主机
1 显示器
2 鼠标
3 键盘
```

如果你只打印 `range`，会出现以下结果：

```
>>> print(range(10))
range(0, 10)
```

`range()` 所返回的对象像一个列表，但实际上却并不是。此对象会在你迭代它时基于所希望的序列返回连续的项，但它没有真正生成列表，这样就能节省内存空间。

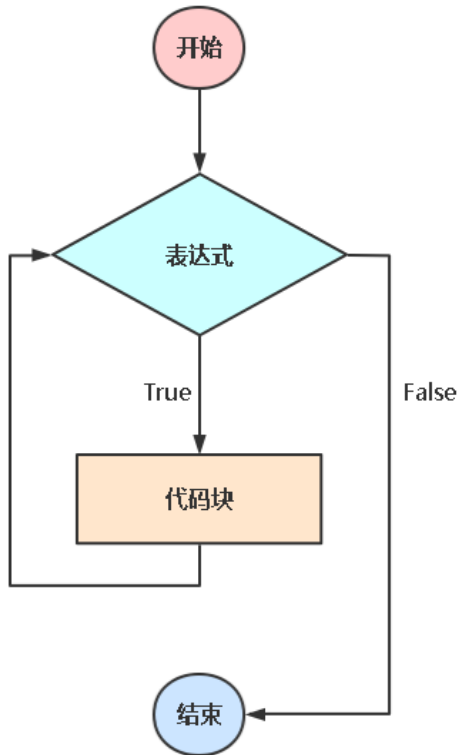
4.3.2 while

while 语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。

执行语句可以是单个语句或语句块。判断条件可以是任何表达式

当判断条件假 false 时，循环结束。

执行流程图如下：



语法格式：

```
while 表达式:  
    代码块
```

1) 输出序列

当条件满足时，停止循环：

```
count = 0  
while count < 5:  
    print(count)  
    count += 1
```

运行结果

```
0  
1  
2  
3  
4
```

2) 死循环

```
count = 0
while True:
    print(count)
    count += 1
# 运行结果
1
2
3
...    # 会一直循环递增数字，直到海枯石烂，天荒地老...
```

注：当表达式值为任何非零、或非空（null）的值均为 true。
示例：

4.3.3 continue 和 break 语句

continue 当满足条件时，跳出本次循环。

break 当满足条件时，跳出所有循环。

只有在 for、while 循环语句中才有效。

示例 1：在循环中，满足条件跳过本次循环

```
for n in range(1,6):
    if n == 3:
        continue
    else:
        print(n)
# 运行结果
1
2
4
5
或者，结果同上
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue
    else:
        print(count)
```

示例 2：在循环中，满足条件终止循环

```
for n in range(1,6):
    if n == 3:
        break
    else:
        print(n)
# 运行结果
1
2
```

或者，结果同上

```
count = 0
while count < 5:
    count += 1
    if count == 3:
        break
    else:
        print(count)
```

示例 3：输入错误次数超过三次退出

一个用户登录场景，输入用户名如果正确“登录成功”退出，否则三次输入错误退出。

```
#!/usr/bin/python
count = 0
while 1:
    if count < 3:
        name = input("请输入你的用户名：")
        if name == "aliang":
            print("登录成功.")
            break
        else:
            print("用户名错误，请重新输入!")
            count += 1
    else:
        print("超出错误次数，退出!")
        break
```

加强版：如果输入为空，则给出提示

```
#!/usr/bin/python
count = 0
while 1:
    if count < 3:
        name = input("请输入你的用户名：").strip() # .strip() 去除首尾空格
        if len(name) == 0:
            print("输入不能为空!")
            continue
        elif name == "aliang":
            print("登录成功.")
            break
        else:
            print("用户名错误，请重新输入!")
            count += 1
    else:
        print("超出错误次数，退出!")
        break
```

4.3.4 循环中的 else 子句

循环语句可能带有 `else` 子句，它会在循环耗尽了可迭代对象（使用 `for`）或循环条件变为假（使用 `while`）时被执行。

示例 1:

```
for n in range(1,7):
    for x in range(4,11):
        if n == x:
            print(n)
            break
    else:
        print("内层循环结束，当前 n 的值: %s" % n)
```

运行结果

内层循环结束，当前 n 的值: 1

内层循环结束，当前 n 的值: 2

内层循环结束，当前 n 的值: 3

4

5

6

外层循环读取序列中 1，内层循环序列元素逐个与 1 判断是否相等，如果相等打印并退出循环，否则执行 `else` 打印 1, 以此类推。

示例 2:

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("end")
```

运行结果

0

1

2

3

4

end

4.2.3 pass 语句

`pass` 是空语句，为了保持程序结构的完整性。例如当你写完语法结构，一时想不起来代码块，可以先用 `pass`，表示不做任何事情。

示例 1:

```
while True:
    pass
```

示例 2:

```
age = int(input("请输入你的年龄："))
if age > 18:
    print("恭喜，你已经成年！")
elif age < 18:
    print("抱歉，你还未成年！")
else:
    pass
```

第五章 Python 文件操作

5.1 open() 函数

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

打开文件并返回文件对象，如果该文件不能被打开，则引发 OSError。

参数如下：

- *file* 将要打开的文件路径
- *mode* 可选字符串，指定打开文件的模式，默认值是 'r'，这意味着以文本模式打开并读取。

字符	含义
'r'	读取（默认）
'w'	写入，并先截断文件
'x'	排它性创建，如果文件已存在则失败
'a'	写入，如果文件存在则在末尾追加
'b'	二进制模式
't'	文本模式（默认）
'+'	打开用于更新（读取与写入）

默认模式为 'r'（打开用于读取文本，与 'rt' 同义）。模式 'w+' 与 'wb+' 将打开文件并清空内容。模式 'r+' 与 'rb+' 将打开文件并不清空内容。

- *buffering* 可选的整数，用于设置缓冲策略。传递 0 以切换缓冲关闭（仅允许在二进制模式下），1 选择行缓冲（仅在文本模式下可用），并且 >1 的整数以指示固定大小的块缓冲区的大小（以字节为单位）。如果没有给出 *buffering* 参数，则默认缓冲策略的工作方式如下：
 - 二进制文件以固定大小的块进行缓冲；使用启发式方法选择缓冲区的大小，尝试确定底层设备的“块大小”或使用 `io.DEFAULT_BUFFER_SIZE`。在许多系统上，缓冲区的长度通常为 4096 或 8192 字节。
 - “交互式”文本文件（`isatty()` 返回 `True` 的文件）使用行缓冲。其他文本文件使用上述策略用于二进制文件。
- *encoding* 用于设置解码或者编码文件的编码的名称。默认编码是依赖于平台的（不管 `locale.getpreferredencoding()` 返回何值）
- *errors* 可选字符串参数，用于指定如何处理编码和解码错误。

- `newline` 控制“通用换行”默认如何生效，它默认是 `None`，`''`，`'\n'`，`'\r'` 和 `'\r\n'`。
- `closefd` 如果是 `False` 并且给出了文件描述符而不是文件名，那么当文件关闭时，底层文件描述符将保持打开状态。如果给出文件名则 `closefd` 必须为 `True`（默认值），否则将引发错误。
- `opener` 自定义开启器

例如：打开一个文件

```
>>> f = open('test.txt', 'r')
```

`open()` 函数打开文件返回一个文件对象，并赋予 `f`，`f` 就拥有了这个文件对象的操作方法，如下：

方法	描述
<code>f.read([size])</code>	读取 <code>size</code> 字节，当未指定或给负值时，读取剩余所有的字节，作为字符串返回
<code>f.readline([size])</code>	从文件中读取下一行，作为字符串返回。如果指定 <code>size</code> 则返回 <code>size</code> 字节
<code>f.readlines([size])</code>	读取 <code>size</code> 字节，当未指定或给负值时，读取剩余所有的字节，作为列表返回
<code>f.write(str)</code>	写字符串到文件
<code>f.flush</code>	刷新缓冲区到磁盘
<code>f.seek(offset[, whence=0])</code>	在文件中移动指针，从 <code>whence</code> （0 代表文件起始位置，默认。1 代表当前位置。2 代表文件末尾）偏移 <code>offset</code> 个字节
<code>f.tell()</code>	当前文件中的位置（指针）
<code>f.close()</code>	关闭文件

5.2 文件对象操作

数据样例：

```
# vim computer.txt
1. 主机
2. 显示器
3. 键盘
```

示例 1：`read()` 读取所有内容

```
f = open("computer.txt", encoding="utf8")
print(f.read())
# 运行结果
1. 主机
2. 显示器
```

3. 键盘

示例 2：指定读取多少字节

```
f = open("computer.txt", encoding="utf8")
print(f.read(4))
# 运行结果
1. 主机
```

示例 3：readline() 读取下一行内容

```
f = open("computer.txt", encoding="utf8")
print(f.readline())
print(f.readline())
# 运行结果
1. 主机

2. 显示器
```

示例 4：readlines() 读取所有内容返回一个列表

```
f = open("computer.txt", encoding="utf8")
print(f.readlines())
['1. 主机\n', '2. 显示器\n', '3. 键盘']
```

示例 5：write() 写入字符串到文件

```
f = open("computer.txt", encoding="utf8", mode="a")
f.write("\n4. 鼠标")
f.flush()
```

示例 6：tell() 获取当前指针的位置

```
f = open("computer.txt", encoding="utf8")
print("当前文件指针的位置：%s" %f.tell())
print("读取一行内容：%s" %f.readline())
print("当前文件指针的位置：%s" %f.tell())
print("读取一行内容：%s" %f.readline())
f.close()
# 运行结果
当前文件指针的位置：0
读取一行内容：1. 主机

当前文件指针的位置：10
读取一行内容：2. 显示器
```

示例 6：seek() 改变指针位置，从指定位置读取

```
f = open("computer.txt", encoding="utf8")
print("当前文件指针的位置：%s" %f.tell())
print("读取一行内容：%s" %f.readline())
print("移动指针到开头")
f.seek(0)
print("当前文件指针的位置：%s" %f.tell())
```



```
print("读取一行内容: %s" %f.readline())
f.close()
# 运行结果
当前文件指针的位置: 0
读取一行内容: 1. 主机

移动指针到开头
当前文件指针的位置: 0
读取一行内容: 1. 主机
```

5.3 文件对象增删改查

在 shell 中，我们要想对文件指定行插入内容、替换等操作，可以使用 sed 工具很容易实现，在 Python 中通过 open() 函数也可以实现类似需求。

主要思路是先读取内容修改，再写会文件，以下举几个常用的情况。

1、在第一行增加一行

例如：在开头添加一个字符串

```
f = open("computer.txt",encoding="utf8")
data = f.read()
data = "0. 电脑\n" + data

f = open("computer.txt",encoding="utf8", mode="w")
f.write(data)
f.flush()
f.close()
```

先将数据读出来保存到变量中，然后把要添加的字符串与原来的数据拼接，最后打开并写入拼接好的数据。

2、在指定行添加一行

例如：在第二行添加一个字符串

```
f = open("computer.txt",encoding="utf8")
data_list = f.readlines()
data_list.insert(1,' 音响\n')
data = ''.join(data_list)

f = open("computer.txt",encoding="utf8", mode="w")
f.write(data)
f.flush()
f.close()
```

先将数据以列表存储，然后根据列表索引插入指定位置（哪一行），再使用 join 把列表拼接成字符串，最后打开文件并写入字符串。

3、在匹配行前一行或后一行添加字符串

```
s = "键盘"
f = open("computer.txt",encoding="utf8")
```

```
data_list = f.readlines()
m_index = data_list.index(list(filter(lambda x:s in x, data_list))[0]) #filter() 过滤
出包含“键盘”的元素并获取索引
data_list.insert(m_index, '麦克风\n') # m_index + 1 就是下一行

f = open("computer.txt", encoding="utf8", mode="w")
f.writelines(data_list)
f.flush()
f.close()
```

这里采用另一种方式 writelines 写入列表数据到文件。

4、删除指定行

例如：删除第三行，与在指定行添加同理

```
del_line = 3 # 删除行号
f = open("computer.txt", encoding="utf8")
data_list = f.readlines()
data_list.pop(del_line-1)

f = open("computer.txt", encoding="utf8", mode="w")
f.writelines(data_list)
f.flush()
f.close()
```

如果只想保留第一行至第三行，可以直接对列表切片 data_list = f.readlines()[0:2]

5、删除匹配行

例如：删除匹配关键字的行

```
del_key = "键盘" # 删除匹配“键盘”的行
f = open("computer.txt", encoding="utf8")
data_list = f.readlines()
data_list = list(filter(lambda x:del_key not in x, data_list)) # filter() 过滤出不包含
“键盘”的行

f = open("computer.txt", encoding="utf8", mode="w")
f.writelines(data_list)
f.flush()
f.close()
```

6、全局替换字符串

```
old_str = "麦克风"
new_str = "耳机"
f = open("computer.txt", encoding="utf8")
data = f.read().replace(old_str, new_str)

f = open("computer.txt", encoding="utf8", mode="w")
f.write(data)
f.flush()
f.close()
```

7、在指定行替换字符串

```
line = 3
old_str = "器"
new_str = "屏"
f = open("computer.txt", encoding="utf8")
data_list = f.readlines()
old_value = data_list[line-1:line] # 获取第三行值, 列表返回
new_value = ''.join(old_value).replace(old_str, new_str) # 将列表转为字符串并替换
data_list[line-1] = new_value # 将第三行值重置

f = open("computer.txt", encoding="utf8", mode="w")
f.writelines(data_list)
f.flush()
f.close()
```

8、实时读取文件新增内容，类似 tail -f

```
f = open("computer.txt", encoding="utf8")
f.seek(0, 2) # 每次打开文件都将文件指针移动到末尾
while True:
    line = f.readline()
    if line:
        print(line)
```

while 是个一直循环读取最新行，比较消耗性能，这时我们可以加个休眠来改善。

例如每秒读取一次：

```
import time
f = open("computer.txt", encoding="utf8")
f.seek(0, 2) # 每次打开文件都将文件指针移动到末尾
while True:
    line = f.readline()
    print(1)
    if line:
        print(line)
    else:
        time.sleep(1)
```

9、处理大文件

在上述方式中，我们一般将数据存放到列表中，实际会存放到内存，如果读取的文件上 G 时，就会导致占用大量内存，甚至内存会爆掉，对于这种情况应这样：

方法 1：open() 打开文件返回的对象本身就是可迭代的，利用 for 循环迭代可提高处理性能

```
f = open("computer.txt", encoding="utf8")
for line in f:
    print(line.strip('\n'))
```

方法 2：每次只读取固定字节

```
f = open("computer.txt", encoding="utf8")
while True:
    data = f.read(1024)
    print(data)
```

```
if not data:
    break
```

5.4 with 语句

在处理一些事务时，可能会出现异常和后续的清理工作，比如读取失败，关闭文件等，Python 对于这种情况提供一种简单的处理方式：with 语句。

```
with open("computer.txt",encoding="utf8") as f:
    data = f.read()
    print(data)
```

这样就不需要明确指定关闭文件了，一些异常、清理的工作都交给 with 处理。
with 表达式其实是 try-finally 的简写形式。

```
f = open('computer.txt',encoding="utf8")
try:
    data = f.read()
finally:
    f.close()
```

第六章 Python 函数

函数是指一段可以直接被另一段程序或代码引用的程序或代码。

一个较大的程序一般应分为若干个程序块，每一个模块用来实现一个特定的功能。

在编写代码时，常将一些常用的功能模块编写成函数，放在函数库中供公共使用，可减少重复编写程序段和简化陈拿高薪结构。

6.1 函数定义与调用

函数语法格式：

```
def functionName(parms1, parms2, ...):
    代码块
    return 表达式
```

关键字 def 定义一个函数，后面跟函数名称和带括号的形式参数列表。

例如定义一个打印功能的函数。

```
def hello():
    print("Hello World!")

hello()
```

定义好函数，如果没有引用是不会执行的，正如上面使用 hello() 表示引用该函数，可以在程序别的地方使用。

如果你学习过其他语言，你可能认为 hello 不是函数而是一个过程，因为它并不返回值。事实上，即使没有 return 语句的函数也会返回一个值，默认是 None。一般来说解释器不会打印它，如果想看到可以使用 print()。

```
print(hello())
```

写一个返回列表（而不是把它打印出来）的函数，非常简单：

```
def hello():
    return "Hello World!"

h = hello()
print(h)
```

正如上述示例，函数通过 return 返回值，当然，print 在调试函数代码时会起到很好的帮助。

6.2 函数参数

6.3.1 接收参数

```
def s(a, b):
    return a + b
print(s(1,2))
# 运行结果
3
```

a 和 b 可以理解为是个变量，由函数里代码块引用。调用函数时，小括号里面的值数量要对应函数参数数量，并根据顺序赋值。如果传入参数数量不一致，会抛出 TypeError 错误。

如果不想一一对应传参，也可以指定对应关系：

```
print(s(b=2, a=1))
```

函数参数也可以是数组：

```
def f(l):
    return l
print(f([1,2,3]))
```

示例：生成列表，根据传入的数值作为结束值

```
def seq(n):
    result = []
    x = 0
    while x < n:
        result.append(x)
        x += 1
    return result
print(seq(9))
# 运行结果
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

6.3.2 参数默认值

参数默认值是指预先给参数定义值，如果调用函数时没指定该值，则用默认值。

```
def s(a, b=2):
    return a + b
print(s(1))
print(s(1, 3))
# 运行结果
3
4
```

6.3.3 接受任意数量参数

上面形式是固定参数数量，当不知道有多少个参数时可以在形参前面加上*与**，表示可接受任意多个参数。

*name 接受任意多个参数，并放到一个元组中。

例如定义个函数，向其传入整数，计算出相加的总和：

```
def func(*seq):
    x = 0
    for n in seq:
        x += n
    return x
print(func(1, 2, 3))
print(func(2, 4, 6))
# 运行结果
6
12
```

**name 接受一个键值，并存储为字典。

例如定义个函数，向其传入键值，并根据字典形式处理打印键值：

```
def func(**computer):
    for k, v in computer.items():
        print("名称:%s\t 价格:%s" % (k, v))
func(主机=5000, 显示器=1000, 鼠标=60, 键盘=150)
# 运行结果
名称:主机      价格:5000
名称:显示器    价格:1000
名称:鼠标      价格:60
名称:键盘      价格:150
```

你也许在查资料的时候，会看到这样写（*args, **kwargs），其实与上面只是名字不一样：

```
def func(a, *args, **kwargs):
    return "%s\n%s\n%s" % (a, args, kwargs)

print(func("hello", 1, 2, 3, 主机=5000, 显示器=1000))
# 运行结果
```

```
hello
(1, 2, 3)
{'主机': 5000, '显示器': 1000}
```

6.3 匿名函数 (Lambada 表达式)

lambda 关键字用于创建一个小的匿名函数，但仅限于单个表达式。

```
s = lambda a, b: a+b
print(s(1,2))
等价于
def func(a, b):
    return a+b
```

也可以增加条件判断：

```
t = lambda x: "整除" if x % 2 == 0 else "不整除"
print(t(6))
print(t(7))
# 运行结果
整除
不整除
```

lambda 与普通函数相比，省去了定义函数的过程和函数名称，让代码更加精简，其他并没有太多作用。如果用 lambda 函数不能使你的代码变得更清晰时，这时你就要考虑使用常规的方式来定义函数。

6.4 作用域

作用域听着挺新鲜，其实也很理解，就是限制一个变量或一段代码可用范围，不在这个范围就不可用。提高了程序逻辑的局部性，减少名字冲突。

Python 的作用域一共有 4 种，分别是：

- **L (Local)**：局部变量，包含在 def 关键字定义的语句块中，即函数中定义的变量。
- **E (Enclosing)**：嵌套作用域，包含了非局部(non-local)也非全局(non-global)的变量。比如两个嵌套函数，一个函数（或类）A 里面又包含了一个函数 B，那么对于 B 中的名称来说 A 中的作用域就为 nonlocal。
- **G (Global)**：全局作用域，当前脚本的最外层，比如当前模块的全局变量。
- **B (Built-in)**：内置作用域，包含了内建的变量/关键字等。最后被搜索



Python 变量查找顺序：局部->全局（global）->内置（built-in）

6.4.1 全局和局部作用域

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

```
a = 2
def func():
    b = 3
print(a)
print(b)
# 运行结果
2
Traceback (most recent call last):
  File "C:/Users/zhenl/Desktop/test.py", line 465, in <module>
    print(b)
NameError: name 'b' is not defined
```

a 变量的作用域是整个代码中有效，称为全局变量，也就是说一段代码最开始定义的变量。
b 变量的作用域在函数内部，也就是局部变量，如果超出使用范围（函数外部），则会报错。。
这么一来，全局变量与局部变量即使名字一样也不冲突。
如果函数内部的变量也能在全局引用，需要使用 global 声明：

```
a = 2
def func():
    global b
    b = 3
print(a)
func() # 需要调用函数再引用变量
print(b)
# 运行结果
2
3
```

使用 global 声明变量后外部是可以调用函数内部的变量。

6.4.2 内置作用域

内置作用域是通过一个名为 `builtin` 的标准模块来实现的，但是这个变量名自身并没有放入内置作用域内，所以必须导入这个文件才能够使用它。在 Python3.0 中，可以使用以下的代码来查看到底预定义了哪些变量：

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

注：Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 if/elif/else/、try/except、for/while 等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问。

6.4.3 嵌套作用域

嵌套作用域是指在一个函数里定义再定义函数，即嵌套函数，内部函数的可以访问外部函数作用域。

```
def outer():
    x = 1
    def inner():
        print(x) # 1
    return inner() # 2
outer()
# 运行结果
1
```

#1 位置引用变量，会先查找局部作用域（inner），如果没有会从外层作用域（outer）查找，即获取到 `x=1`，所以这里 inner 函数可以访问外层作用域。

#2 位置返回 inner 函数执行的结果。

6.6 闭包

闭包本身是一个晦涩难懂的概念，这里不再详解，你可以简单粗暴地理解为**闭包就是一个定义在函数内部的函数**，闭包使得变量即使脱离了该函数的作用域范围也依然能被访问到。像上面所说的嵌套函数也是**闭包**的一种形式（将内部嵌套定义的函数作为返回值）。

```
def outer():
    x = 1
    def inner():
        print(x) # 1
    return inner # 不加括号，表示返回函数体
sf = outer()
sf() # 调用，等同于 outer() ()
# 运行结果
1
```

在一个外部函数内定义了一个函数，内部函数里引用外部函数的变量，并且外部函数的返回值是内部函数的引用，这样就构成了一个闭包，并且满足了闭包的三个条件：

- 两层以上嵌套关系

- 内部函数调用外部函数定义的变量
- 外部函数返回内部函数体对象, 而不是函数体结果（加括号）

6.8 函数装饰器

6.8.1 无参数装饰器

装饰器（decorator）本身是一个函数，用于包装另一个函数或类，它可以让其他函数在不需要改动代码情况下动态增加功能，这样有助于让我们的代码更简短。

有这么一个打印的函数：

```
def hello():  
    print("我是原函数")
```

现在希望能给这个函数添加记录日志。

```
def decorator(func):  
    def f():  
        print("原函数开始执行了")  
        func()  
        print("原函数执行结束了")  
    return f
```

定义一个嵌套函数，形参接收一个函数，内部函数读取形参值并执行，然后外部函数返回函数体。用一个变量来接收：

```
dec = decorator(hello)  
dec()  
# 运行结果  
原函数开始执行了  
我是原函数  
原函数执行结束了
```

装饰器帮我们完成打印函数执行之前和之后处理。

执行顺序：

第一步：@语法糖等同于 `dec = decorator(hello)`

第二步：执行装饰器 `decorator(hello)` -> 内函数接收访问外函数传入的 `func` 函数 -> 代码块处理 `func` 函数 -> `return` 返回函数体

第三步：`dec()` 调用内函数获取结果

在实际应用中，我们每一个需要调用 `decorator` 装饰器的地方都需要这么写，也显得比较麻烦，Python 提供了一个更简洁的引用方式语法糖 “@”

```
@decorator  
def hello():  
    print("我是原函数")  
  
hello()
```

6.8.2 带参数装饰器

如果给一个带参数的函数加装饰器该怎么做呢，先试试：

```
@decorator
def hello(msg):
    print(msg)

hello("我是原函数")
# 运行结果
TypeError: f() takes 0 positional arguments but 1 was given
```

类型错误这是为什么呢？因为现有的装饰器是不接受其他参数的，只接受函数。如果你需要传参，还影响内部函数指定参数：

```
def decorator(func):
    def f(msg):
        print("原函数开始执行了")
        func(msg)
        print("原函数执行结束了")
    return f

@decorator
def hello(msg):
    print(msg)

hello("我是原函数")
```

接收不固定参数也一样：

```
def decorator(func):
    def f(*args, **kwargs):
        print("原函数开始执行了")
        func(*args, **kwargs)
        print("原函数执行结束了")
    return f

@decorator
def hello(*msg, **kwargs):
    print(msg, kwargs)

hello("我是原函数", 你好="hello")
# 运行结果
原函数开始执行了
('我是原函数',) {'你好': 'hello'}
原函数执行结束了
```

第七章 Python 常用内建函数

Python 解释器内置了很多函数和类型，您可以在任何时候使用它们。

在前面我们已经学习到了 `print()`、`len()`、`open()`、`range()`、`join()` 等这些都是你内建函数，你可以通过查看官方文档（<https://docs.python.org/zh-cn/3.8/library/functions.html>）或者 `builtin` 方法获取内置的函数：

```
>>> import __builtin__
>>> dir(__builtin__)
```

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

7.1 高阶函数

高阶函数是至少满足这两个任意中的一个条件：

- 1) 能接受一个或多个函数作为输入。
- 2) 输出一个函数。

所以前面将的闭包、装饰器是属于高阶函数。

在 Python 内置函数里面也有高阶函数，例如 `map()`、`filter()`

7.1.1 map()

语法：map(*function*, *iterable*, ...)

map 函数接收两个参数，一个是函数，一个是可迭代对象（例如 list、dict、str 等），其中函数是用于处理可迭代对象中每一个元素，处理完后返回一个迭代器对象。

如果传入额外的 iterable 参数，function 必须接受相同个数的形参并应用于从所有可迭代对象中获取的项。

在 Python2 版本中 map() 返回的是一个列表，Python3 版本返回是一个迭代器，这样好处是：

- 列表节省内存，迭代器在内存中只占一个数据的空间，因为每次取值都是上一条数据释放后才加载当前数据。而列表则是一次获取所有值。
- 提供一个通用不依赖索引的迭代取值方式。

示例 1：把列表中元素都乘以 2，可以利用 map() 函数完成处理

```
num = range(1, 11)

def handle(n):
    return n * 2
result = map(handle, num)
print(list(result))
# 运行结果
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

map() 第一个参数接收 handle 函数，第二个参数接收 num 列表，handle 函数接收 num 每个元素进行处理，列表处理完成后 map 返回一个迭代对象，使用 list 函数对其转换打印出列表。也可以使用匿名函数完成相同效果：

```
result = map(lambda n:n * 2, num)
print(list(result))
```

示例 2：给列表中元素加个后缀

```
result = map(lambda n:str(n) + ".txt", num)
print(list(result))
# 运行结果
['1.txt', '2.txt', '3.txt', '4.txt', '5.txt', '6.txt', '7.txt', '8.txt', '9.txt', '10.txt']
```

接下来利用前面所学的函数知识写一个类似 map() 函数功能，加深印象。

```
def handle(n):
    return n * 2
def map_test(func, iter):
    l = []
    for i in iter:
        x = func(i)
        l.append(x)
    return l.__iter__() # 增加 iter 方法，将列表转为迭代器
result = map_test(handle, num)
print(list(result))
```

总结：map() 函数可以对序列中每个值进行某种批量转化操作，然后将结果作为迭代器 (Iterator) 返回，迭代器可以利用 for 循环或者 next() 函数来访问每个值。

7.1.2 filter()

语法：filter(*function*, *iterable*)

filter 函数与 map 函数类似，也是接收两个参数，并返回一个迭代器。但如果 function 返回是 None，则会将元素移除。

示例 1：过滤列表中的奇数（与 2 整除）

```
num = range(1, 11)
def handle(n):
    if n % 2 == 0:
        return n

result = filter(handle, num)
print(list(result))
# 运行结果
[2, 4, 6, 8, 10]
```

或者使用匿名函数：

```
result = filter(lambda n: n % 2 == 0, num)
print(list(result))
```

示例 2：过滤出大于 5 的元素

```
result = filter(lambda n: n > 5, num)
print(list(result))
# 运行结果
[6, 7, 8, 9, 10]
```

7.2 排序函数

7.2.1 sorted()

语法：sorted(*iterable*, *, *key=None*, *reverse=False*)

根据 iterable 中的项返回一个新的已排序列表。

具有两个可选参数：

- key 指定带有单个参数的函数，用于从 iterable 的每个元素中取出比较的键（例如 key=str.lower），默认值为 None（直接比较元素）
- reverse 为一个布尔值，如果设置为 True，则每个列表元素将按反向顺序比较进行排序。

示例 1：对列表排序

```
num = [2, 3, 4, 1, 5]
a = ["b", "c", "a"]
print(sorted(num))
str1 = ["b", "c", "a"]
print(sorted((str1)))
# 运行结果
[1, 2, 3, 4, 5]
['a', 'b', 'c']
```

示例 2：对字典中的值排序

```
dict = {'a':86, 'b':23, 'c':45}
result = sorted(dict.items(), key=lambda x:x[1])
print(result)
# 运行结果
[('b', 23), ('c', 45), ('a', 86)]
```

7.2.3 reversed()

语法：reversed(*seq*)

返回一个反向的 iterable（注意不是排序）。

示例：

```
num = [1, 2, 3]
print(list(reversed(num)))
str1 = ["a", "b", "c"]
print(list(reversed(str1)))
```

7.3 最小最大值

7.3.1 min()

语法 1：min(iterable, *, key, default)) 返回可迭代对象中最小的元素。

语法 2：min(arg1, arg2, *args[, key]) 返回位置参数最小的元素，如果提供了两个及以上的位置参数，则返回最小的位置参数。

示例：

```
num = [1, 2, 3]
print(min(num))
print(min("a", "b", "c"))
# 运行结果
1
a
```

7.3.2 max()

用法与 min(*) 一样，只是返回的是最大的元素。

```
num = [1, 2, 3]
print(min(num))
print(min("a", "b", "c"))
# 运行结果
3
c
```

7.4 求和

语法：sum(iterable, /, start=0)

从 start 开始自左向右对 iterable 的项求和并返回总计值。iterable 的项通常为数字，而 start 值则不允许为字符串。

示例：

```
num = [1, 2, 3]
print(sum(num))
# 运行结果
6
```

7.4 可迭代对象计数

语法：enumerate(iterable, start=0)

返回一个枚举对象。iterable 是一个可迭代对象，start 设置计数值，默认为 0。

示例：

```
computer = ["主机", "显示器", "鼠标", "键盘"]
print(list(enumerate(computer)))
print(list(enumerate(computer, start=1)))
# 运行结果
[(0, '主机'), (1, '显示器'), (2, '鼠标'), (3, '键盘')]
[(1, '主机'), (2, '显示器'), (3, '鼠标'), (4, '键盘')]
```

示例 2：

```
result = enumerate(computer, start=1)
for k, v in result:
    print("索引: %s, 键: %s" % (k, v))
# 运行结果
索引: 1, 键: 主机
索引: 2, 键: 显示器
索引: 3, 键: 鼠标
索引: 4, 键: 键盘
```


7.5 多个迭代对象聚合

语法: `zip(*iterables)`

将多个可迭代对象创建一个聚合，返回一个元组的迭代器。

如果多个可迭代对象长度不一样，当最短可迭代对象被耗尽时，迭代器将停止迭代。

当只有一个可迭代对象参数时，它将返回一个单元组的迭代器。

示例：

```
x = [1, 2, 3]
y = [4, 5, 6]
zipped = zip(x, y)
print(list(zipped))
# 运行结果
[(1, 4), (2, 5), (3, 6)]
```

7.6 字符串转换表达式 `eval()`

`eval(expression[, globals[, locals]])`

实参是一个字符串，以及可选的 `globals` 和 `locals`。`globals` 实参必须是一个字典。`locals` 可以是任何映射对象。

把字符串当成 Python 表达式处理并返回计算结果：

```
x = 1
print(eval('x+1'))
# 运行结果
2
```

7.7 获取当前所有变量

`globals()` 字典格式返回当前范围的全局变量。允许修改原变量的值

`locals()` 字典格式返回当前范围的局部变量。返回的是对原来变量的拷贝，不允许修改原变量值

```
a = 1
def f():
    b = 2
    print("局部变量: %s" % locals())
print("全局变量: %s" % globals())
f()
# 运行结果
全局变量: {'__name__': '__main__', '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x000001E7D7CC1CF8>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__cached__': None, 'a': 1}
局部变量: {'b': 2}
```

第八章 Python 类（面向对象编程）

类是对现实生活中一类具有共同特征的事物的抽象描述，使得对更易于编码和设计。

理解面向对象编程得先理解类、对象以及他们之间关系。

- 类：共同特征的事物的抽象描述，通俗的讲就是分类，例如人类、动物类、电脑类、手机类等
- 对象：类的实体，实际存在的事物，例如电脑类的“主机”、“显示器”
- 类与对象的关系：类是由对象来的，这个过程叫做抽象化。对象是由类中的某个一个事物，这个过程叫做实例化。

其他特点：

- 封装：把相同功能的属性、方法封装到类中，不允许有其他类的东西，例如人两条腿走路，小狗四条腿走路，这样是不能封装到一个类中
- 方法：类里面的函数，也称为成员函数
- 属性：变量
- 实例化：创建一个类的具体实例对象
- 继承：类之间的关系，比如猫狗是一类，他们都有四条腿，狗继承了这个四条腿，拥有了这个属性。

什么是面向对象编程？

面向对象编程（Object Oriented Programming, OOP，面向对象程序设计）是一种计算机编程架构。Python 就是这种编程语言。

8.1 类的定义

语法格式：

```
# class 关键字定义个类，
class ClassName():
    # 类中的函数，称为方法
    def funcName(self):
        pass
```

self 代表类本身。类中的所有的函数的第一个参数必须是 self。

例如我们定义一个电脑类，为更好理解，以下采用伪代码形式：

电脑（类）：

特征（属性）：主机，显示器，键盘，鼠标…

host = "4C8G"

displayer = "27 寸"

keyboard = "机械键盘"

mouse = "无线鼠标"

功能（方法）：办公，上网，玩游戏，看电影…

def office():

 办公

def internet():

```
    上网
def game():
    玩游戏
...
```

空调（类）：

特征（属性）：品牌，功耗，冷暖类型，变频...

```
    brand = "品牌"
    power = "功耗"
```

功能（方法）：制冷，制热，除湿...

```
    def cool():
        制冷
    def hot():
        制热
    def wet():
        除湿
    ...
```

人（类）：

特征（属性）：姓名，性别，年龄，身高...

```
    name = "阿良"
    sex = "男"
    age = "30"
```

功能（方法）：工作，学习，生孩子...

```
    def office():
        工作
    def study():
        学习
    def wet():
        除湿
    ...
```

属性：相当于全局变量，实例对象共有的属性。

方法：定义类的函数操作。这些方法可以通过类或者实例化后的对象进行调用。

8.2 类的书写规范

1、使用 class 关键字声明一个类

2、命名规则

类名称一般采用大驼峰命名，例如 LiZhenLiang

3、类注释

类添加注释，用于说明该类的用途，提高可阅读性：

```
class MyClass():
    """
    这是一个测试类
    """

print(MyClass.__doc__)
help(MyClass)
# 运行结果
    这是一个测试类

Help on class MyClass in module __main__:

class MyClass(builtins.object)
 |   这是一个测试类
 |   ...
```

4、类中只能存在两种数据：属性和方法

5、一般创建类会带有特定的初始状态

使用__init__作为初始化方法，可以用来设置对象属性，并给予初始值，可以是参数或者固定值：

```
class MyClass():
    """
    这是一个测试类
    """

    def __init__(self):
        self.host = "4C8G"
        self.display = "27 寸"
        self.keyboard = "机械键盘"
        self.mouse = "无线鼠标"
```

__init__函数定义到类的开头，self.name 变量是一个实例属性，只能在类方法中使用，引用时也要这样 self.name。

init()方法主要作用：

- 在对象生命周期中初始化是最重要的一步；每个对象必须正确初始化后才能正常工作。
- init()参数值可以有多种形式。

init()是一个函数，只不过在类中有一点特殊的作用罢了，每个类，首先要运行它，它规定了类的基本结构。

6、声明属性时必须赋值

7、声明方法时，第一个参数必须是 self，其他按照函数声明规范即可

8.3 类实例化

例如定义一个电脑类。

```
class Computer():
    """
    电脑类
    """
```

```

'''
# 特征
def __init__(self):
    self.host = "4C8G"
    self.displayer = "27 寸"
    self.keyboard = "机械键盘"
    self.mouse = "无线鼠标"
# 方法
def office(self):
    return "办公"
def internet(self):
    return "上网"
def game(self):
    return "玩游戏"

# 类的实例化对象
pc = Computer()
# 查看类的属性
print(pc.host)
# 调用类方法
print(pc.office())

```

运行结果：

```

4C8G
办公

```

当一个类定义了__init__()方法时，类的实例化操作会自动为新创建的类实例发起调用__init__()。因此在这个示例中，可以通过以下语句获得一个经初始化的新实例：

```
pc = Computer()
```

当然，__init__()方法还可以有额外参数以实现更高灵活性。在这种情况下，提供给类实例化运算符的参数将被传递给__init__()。例如：

```

class Computer():
    '''
    电脑类
    '''
    # 特征
    def __init__(self, name):
        self.host = "4C8G"
        self.displayer = "27 寸"
        self.keyboard = "机械键盘"
        self.mouse = "无线鼠标"
        self.name = name
    # 方法
    def office(self):
        return "%s 办公，电脑配置：%s" %(self.name, self.host)
    def internet(self):
        return "上网"

```

```
def game(self):
    return "玩游戏"

# 类的实例化对象，将对象赋值 pc
pc = Computer("阿良")
## 查看类的属性
# print(pc.host)
# 调用类方法
print(pc.office())
```

运行结果：

阿良 办公，电脑配置：4C8G

类方法调用：

- 1) 类方法之间调用：self.<方法名>（参数），参数不需要加 self
- 2) 外部调用：<实例名>.<方法名>

8.4 属性操作

8.4.1 访问、修改、添加、删除

实例化后可以动态操作类属性：

```
# 访问
print(pc.host) # 属性引用
# 添加
pc.sound = "音响"
print(pc.sound)
# 修改
print(pc.host)
pc.host = "8C16G" # 变量重新赋值
print(pc.host)
# 删除
del pc.mouse
print(pc.mouse)
```

运行结果：

```
4C8G
音响
4C8G
8C16G
AttributeError: 'Computer' object has no attribute 'mouse'
```

8.4.2 属性私有化

为了更好的保护属性安全，即不能随意修改，一般会将属性定义为私有属性。

1、单下划线

实现模块级别的私有化，以单下划线开头的变量和函数只能类或子类才能访问。当 `from modulename import *` 时将不会引入以单下划线开头的变量和函数。

```
class Computer():
    """
    电脑类
    """
    # 特征
    def __init__(self, name):
        self.host = "4C8G"
        self.display = "27 寸"
        self.keyboard = "机械键盘"
        self.mouse = "无线鼠标"
        self._name = name
    # 方法
    def office(self):
        return "%s 办公, 电脑配置: %s" % (self._name, self.host)
    def internet(self):
        return "上网"
    def game(self):
        return "玩游戏"

# 类的实例化对象, 将对象赋值 pc
pc = Computer("阿良")
# 引用属性
print(pc._name)
```

`self._name` 变量其实就是做了个声明, 说明这是个内部变量, 外部不要去引用它。

2、双下划线

以双下划线开头的变量, 表示私有变量, 受保护的, 只能类本身能访问, 连子类也不能访问。避免子类与父类同名属性冲突。

在单下划线基础上又加了一个下划线 `self.__name = name`

```
print(pc.__name)
AttributeError: 'Computer' object has no attribute '__name'
```

提示没有属性, 可见双下划线变量只能本身能用。

如果真想访问私有属性, 可以这样引用:

```
print(pc._Computer__name)
```

`self.__name` 变量编译成了 `self._MyClass__name`, 以达到不能被外部访问的目的, 实际并没有真正意义上的私有。

8.4.3 内置属性（首尾双下划线）

一般保存对象的元数据, 比如

- `__doc__` 类的文档
- `__dict__` 查看类的成员

- `__name__` 类名
- `__module__` 类定义的所在的模块

```
print("__doc__： %s" %Computer.__doc__)
print("__dict__： %s" %Computer.__dict__)
print("__name__： %s" %Computer.__name__)
print("__module__： %s" %Computer.__module__)
```

运行结果

电脑类

```
__dict__: {'__module__': '__main__', '__doc__': '\n  电脑类\n  ', 'host': '4C8G', 'displayer': '27 寸', 'keyboard': '机械键盘', 'mouse': '无线鼠标', 'office': <function Computer.office at 0x0000029029DBBEA0>, 'internet': <function Computer.internet at 0x0000029029DC3400>, 'game': <function Computer.game at 0x0000029029DC3378>, '__dict__': <attribute '__dict__' of 'Computer' objects>, '__weakref__': <attribute '__weakref__' of 'Computer' objects>}
```

```
__name__: Computer
__module__: __main__
```

`dic()` 返回对象内变量、方法

```
>>> dir(MyClass)
['__doc__', '__module__']
```

`dir()` 函数，不带参数时，返回当前范围内的变量、方法的列表。带参数时，返回参数的属性、方法的列表。

8.5 方法

添加一个方法：

```
class Computer():
    ...
    def count(self, x, y):
        return x * y
pc = Computer("阿良")
print(pc.count(6, 6))
```

运行结果

36

8.6 类的继承

例如电脑类，电脑下分为笔记本、台式机、平板等，这些都具有电脑类共同的特征，例如 CPU、内存、硬盘、显示器、键盘、鼠标等。其实子类就是指的不同展现形式的电脑，子类获取父类的特征，这就是继承。

俗话说“龙生龙凤生凤老鼠生子打洞”，孩子继续父类的基因和特征。

对于编程来说，子类继承父类，子类将继承父类的所有方法和属性，这样好处是可以简化代码，提供代码可维护性。

1) 简单继承

```
class Computer():
    '电脑类'
    def __init__(self):
        self.host = "4C8G"
    def office(self):
        return "办公，我的电脑配置：%s" %self.host

class Notebook(Computer):
    '笔记本类'
    pass

class Desktop(Computer):
    '台式机类'
    pass

notebook_pc = Notebook()
print(notebook_pc.office())
# 运行结果
办公，我的电脑配置：4C8G
```

在 Notebook 类中并没有 office 方法，依然可以使用，这就是继承。

重写父类方法：

```
class Computer():
    '电脑类'
    def __init__(self):
        self.host = "4C8G"
    def office(self):
        return "办公，我的电脑配置：%s" %self.host

class Notebook(Computer):
    '笔记本类'
    def __init__(self):
        self.host = "8C16G"
    """
    重写父类方法
    子类方法名、参数应该和父类的方法一样
    """
    def office(self):
        return "休息， 我的电脑配置：%s" %self.host

class Desktop(Computer):
    '台式机类'
    pass
```

```
notebook_pc = Notebook()
print(notebook_pc.office())
# 运行结果
休息， 我的电脑配置：8C16G
```

这里面__init__子类是重复定义了父类的，如果大量调用子类，这段也会多消耗一些资源。所以我们不希望子类重复引用，这种写法看着也有点累赘。我们该怎么做呢，

其实有个一个方法 super() 函数是在父类当中初始化了某些方法，在父类中不用再重复初始化。

使用 super() 函数可以继承父类同名方法：

```
class Notebook(Computer):
    '笔记本类'
    def __init__(self):
        super().__init__()
```

如果重名，子类会覆盖父类方法。一旦子类重新定义构造函数，子类会调用自己的构造方法。

子类增加方法：

```
class Notebook(Computer):
    '笔记本类'
    def __init__(self):
        super().__init__()
    """
    重写父类方法
    子类方法名、参数应该和父类的方法一样
    """
    def office(self):
        super().office()
        return "休息， 我的电脑配置：%s" %self.host
    def internet(self):
        return "上网"
```

```
class Desktop(Computer):
    '台式机类'
    pass
```

```
notebook_pc = Notebook()
print(notebook_pc.office())
print(notebook_pc.internet())
```

8.7 内置函数访问对象属性

8.7.1 getattr()

getattr() 函数用于返回一个对象的属性。

语法：getattr(object, name[, default])

```
class Computer():
    '电脑类'
    def __init__(self, keyboard="鼠标"):
        self.host = "4C8G"
        self.displayer = "27 寸"
        self.keyboard = "机械键盘"
        self.mouse = "无线鼠标"
    def office(self):
        return "办公，电脑配置： %s" %self.host

pc = Computer()
print(getattr(pc, "host"))
print(getattr(pc, "name", "对象没有该属性")) # 如果属性不存在，返回默认值
# 运行结果
4C8G
对象没有该属性
```

8.7.2 hasattr()

hasattr()函数用于判断对象是否具有属性，返回一个布尔值。

语法：hasattr(object, name)

```
print(hasattr(pc, "host"))
print(hasattr(pc, "name"))
```

8.7.3 setattr()

setattr()函数用于给对象重新赋值或添加属性。如果属性不存在则添加，否则重新赋值。

语法：setattr(object, name, value)

```
setattr(pc, "name", "戴尔")
print(hasattr(pc, "name"))
```

8.7.4 delattr()

delattr()函数用于删除对象属性。

语法：delattr(object, name)

```
print(hasattr(pc, "host"))
print(delattr(pc, "host"))
print(hasattr(pc, "host"))
```

8.8 类装饰器

8.8.1 自定义装饰器

与函数装饰器类似，不同的是类要当做函数一样调用：
定义一个函数：

```
def f1():  
    return "Hello World!"  
  
print(f1())
```

如果我们想给 f1 函数添加计时功能，除了函数装饰器，类也可以定义装饰器。

```
import time  
class Decorator():  
    def __init__(self, func):  
        self._func = func  
        self._func_name = func.__name__  
    def __call__(self, *args, **kwargs):  
        start = time.time()  
        self._func() # 执行接收的函数  
        end = time.time()  
        return f"函数的运行时间为: {str(end-start)}"  
@Decorator # Decorator(f1)  
def f1():  
    return "Hello World!"  
  
print(f1())  
  
# 运行结果  
函数的运行时间为: 0.0
```

当我们调用装饰器的时候，它会执行定义的__call__魔法方法，所以也就完成了一个装饰器的定义和调用

__call__方法:可以让类中的方法像函数一样调用。

```
class A():  
    def __call__(self, *args, **kwargs):  
        print(args, kwargs)  
t = A()  
t(1, 2, 3, a=1, b=2, c=3)  
# 运行结果  
(1, 2, 3) {'a': 1, 'b': 2, 'c': 3}
```

8.8.2 类内置装饰器

1、@property

@property：属性装饰器，是把类中的方法当做属性来访问。
在没使用属性装饰器时，类方法是怎样被调用的：

```
class Computer():
    def __init__(self):
        self.host = "4C8G"
        self.display = "27 寸"
    def office(self):
        return "办公，我的电脑配置： %s" %self.host

pc = Computer()
print(pc.office()) # 加小括号
```

使用属性装饰器就可以像属性那样访问了：

```
class Computer():
    def __init__(self):
        self.host = "4C8G"
        self.display = "27 寸"
    @property
    def office(self):
        return "办公，我的电脑配置： %s" %self.host

pc = Computer()
print(pc.office) # 不加小括号一样打印
```

加了@property 之后，类方法就不能再用 pc.office() 方法。

2、@staticmethod

@staticmethod：静态方法装饰器，可以通过类对象访问方法，也可以通过实例化后类对象实例访问。

实例方法的第一个参数是 self，表示是该类的一个实例，称为类对象实例。

而使用静态方法装饰器，类方法第一个参数就不用传入实例本身（self），那么这个方法当做类对象，由 Python 自身处理。

```
class Computer():
    def __init__(self):
        self.host = "4C8G"
        self.display = "27 寸"
    def office(self):
        return "不是静态方法"
    @staticmethod
    def internet():
        return "使用静态方法"

pc = Computer()
print(pc.office())
print(Computer.internet()) # 可以通过类调用静态方法
print(pc.internet()) # 还可以通过类实例化对象访问
```

运行结果

不是静态方法
使用静态方法
使用静态方法

静态方法与普通函数调用方式一样，只不过命名空间是在类里面，必须通过类来调用。

3、@classmethod

@classmethod：类方法装饰器，与静态方法装饰器类似，可以直接通过类访问方法，不需要提前实例化。主要区别在于类方法的第一个参数要传入类对象（cls）：

```
class Computer():
    def __init__(self):
        self.host = "4C8G"
        self.displayer = "27 寸"

    def office(self):
        return "办公"

    @classmethod
    def internet(cls):
        return "类方法"

print(Computer.internet())
```

第九章 Python 异常处理

什么是异常？

顾名思义，异常就是程序因为某种原因无法正常工作了，比如缩进错误、缺少软件包、环境错误、连接超时等等都会引发异常。一个健壮的程序应该把所能预知的异常都应做相应的处理，应对一些简单的异常情况，使得更好的保证程序长时间运行。即使出了问题，也可让维护者一眼看出问题所在。因此本章节讲解的就是怎么处理异常，让你的程序更加健壮。

9.1 捕捉异常语法

```
try:
    expression
except [Except Type]:
    expression
```

try:

执行代码

except:

发生异常时执行的代码

如果在执行 try 块里的业务逻辑代码时出现异常，系统自动生成一个异常对象，该异常对象被提交给 Python 解释器，这个过程被称为引发异常。

当 Python 解释器收到异常对象时，会寻找能处理该异常对象的 except 块，如果找到合适的 except 块，则把该异常对象交给该 except 块处理，这个过程被称为捕获异常。如果 Python 解释器找不到捕获异常的 except 块，则运行时环境终止，Python 解释器也将退出。

9.2 异常类型

常见的异常类型：

异常类型	用途
SyntaxError	语法错误
IndentationError	缩进错误
TypeError	对象类型与要求不符合
ImportError	模块或包导入错误；一般路径或名称错误
KeyError	字典里面不存在的键
NameError	变量不存在
IndexError	下标超出序列范围
IOError	输入/输出异常；一般是无法打开文件
AttributeError	对象里没有属性
KeyboardInterrupt	键盘接受到 Ctrl+C
Exception	通用的异常类型；一般会捕捉所有异常
UnicodeEncodeError	编码错误

内置异常的分类层级结构如下：

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
```

```
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
|   +-- ModuleNotFoundError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- MemoryError
+-- NameError
|   +-- UnboundLocalError
+-- OSError
|   +-- BlockingIOError
|   +-- ChildProcessError
|   +-- ConnectionError
|       |   +-- BrokenPipeError
|       |   +-- ConnectionAbortedError
|       |   +-- ConnectionRefusedError
|       |   +-- ConnectionResetError
|   +-- FileExistsError
|   +-- FileNotFoundError
|   +-- InterruptedError
|   +-- IsADirectoryError
|   +-- NotADirectoryError
|   +-- PermissionError
|   +-- ProcessLookupError
|   +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
```



```
+++ ImportWarning
+++ UnicodeWarning
+++ BytesWarning
+++ ResourceWarning
```

9.3 异常处理

示例 1：打印一个没有定义的变量，名称错误

```
print(name)
# 运行结果
NameError: name 'name' is not defined
```

异常处理：

```
try:
    print(name)
except NameError:
    print("发生名称错误时，执行的代码")
```

示例 2：字符串与整数计算，类型错误

```
'2' + 2
# 运行结果
TypeError: can only concatenate str (not "int") to str
```

异常处理：

```
try:
    n = '2' + 2
except TypeError:
    print("发生类型错误时，执行的代码")
```

但如果是语法错误，是无法通过 try..except 进行处理，例如：

```
while True print('Hello world')
# 运行结果
File "C:/Users/zhenl/Desktop/test.py", line 844
    while True print('Hello world')
                  ^
SyntaxError: invalid syntax
```

解释器会输出语法错误的行号和文件名，以便让你知道去哪里检查。

在上面示例中，我们通过指定异常类型准确的处理，但在实际开发中，往往不知道会引发什么样的异常类型，这时就可以使用 Exception 类型来捕捉所有的异常。

```
try:
    n = '2' + 2
except Exception:
    print("发生错误执行的代码")
```

运行结果

发生错误执行的代码

这样不管是名称错误还是类型错误都可以捕获到进行处理。
在处理异常的同时，也希望将执行的错误信息打印出来，以便让用户检查。
可以将错误输出通过 `as` 关键字保存到变量中，再进行打印：

```
try:
    print(name)
except Exception as e:
    print("错误: %s" %e)
    print("发生错误执行的代码")
# 运行结果
错误: name 'var' is not defined
发生错误执行的代码
```

注：except 也可以不指定异常类型，那么会忽略所有的异常类，这样做有风险，它同样会捕捉 Ctrl+C、sys.exit 等的操作，所以使用 `except Exception` 更好些。

9.4 else 和 finally 语句

9.4.1 else 语句

在异常处理流程中还可以添加一个 else 块，当 try 块没有出现异常时，程序会执行 else 块。

```
name = "aliang"
try:
    print(name)
except Exception as e:
    print("错误: %s" %e)
    print("发生错误执行的代码")
else:
    print("没有出现异常执行的代码")
# 运行结果
aliang
没有出现异常
```

9.4.2 finally 语句

当 try 块不管是异常，都会执行 finally。一般用于业务逻辑处理完后的清理工作，例如打开一个文件，不管是否操作成功，都应该关闭文件。

```
f = open("computer.txt", encoding="utf8")
try:
    data = f.read()
except Exception as e:
    print("错误: %s" %e)
    print("发生错误执行的代码")
```

```
finally:
    f.close()
```

9.4.3 try...except...else...finally

这是一个完整的语句，当一起使用时，使异常处理更加灵活。

```
try:
    print(name)
except Exception as e:
    print("错误: %s" %e)
    print("发生错误执行的代码")
else:
    print("没有出现异常执行的代码")
finally:
    print("最后执行的代码")
```

注：它们语句的顺序必须是 try...except...else...finally，否则语法错误！里面 else 和 finally 是可选的。

9.5 raise 自定义异常

raise 语句允许程序员强制发生指定的异常。用来手动抛出一个异常，使用方法：

语法：raise [exceptionName [(reason)]]

其中，用 [] 括起来的为可选参数，其作用是指定抛出的异常名称，以及异常信息的相关描述。如果可选参数全部省略，则 raise 会把当前错误原样抛出；如果仅省略 (reason)，则在抛出异常时，将不附带任何的异常描述信息。

也就是说，raise 语句有如下三种常用的用法：

- raise: 单独一个 raise。该语句引发当前上下文中捕获的异常（比如在 except 块中），或默认引发 RuntimeError 异常。
- raise 异常类名称: raise 后带一个异常类名称，表示引发执行类型的异常。
- raise 异常类名称(描述信息): 在引发指定类型的异常的同时，附带异常的描述信息。

```
try:
    raise
except Exception as e:
    print("自定义异常类错误: %s" %e)
```

raise 唯一的参数就是要抛出的异常。这个参数必须是一个异常实例或者是一个异常类（Exception 子类）

例如：抛出一个指定的异常

```
name = "aliang"
try:
    print(name)
    raise NameError("程序猿抛出的异常")
```

```
except Exception as e:
    print("错误: %s" %e)
    print("发生错误执行的代码")
else:
    print("没有出现异常执行的代码")
finally:
    print("最后执行的代码")
```

raise 参数必须是一个异常的实例或 Exception 子类。

上面用的 Exception 子类，那么我定义一个异常的实例，需要继承 Exception 类：

```
class MyError(Exception):
    def __init__(self, msg):
        self.msg = msg
    def __str__(self): # __str__自动打印这个方法 return 的数据
        return self.msg

try:
    if 1 != 2:
        raise MyError("手动引发异常")
except MyError as e:
    print("自定义异常类错误: %s" %e)
```

运行结果

自定义异常类错误: 手动引发异常

第十章 Python 自定义模块及导入方法

模块是一系列常用功能的集合体，一个 py 文件就是一个模块。这样做的目的是为了将重复使用的代码有组织的存放一起，方便管理，谁用谁拿。拿的这个过程叫做导入（import）。

导入模块的一般写法：

```
import 模块名称
from 模块名称 import 方法名
```

10.1 自定义模块

既然一个 py 文件都可以作为模块导入，下面定义一个简单的模块（mymodule.py）：

```
#!/usr/bin/python
def count(a, b):
    result = a * b
    return f"{a} 与 {b} 的乘积是: {result}"

class Count():
    def __init__(self, a, b):
```

```
self.a = a
self.b = b
def count(self):
    result = self.a * self.b
    return f"{self.a}与{self.b}的乘积是: {result}"
```

可以看到，在 mymodule.py 中写了一个函数和一个类，该文件就可以作为一个模块被别的 Python 程序使用。

但通常情况下，为了检查代码正确性，会进行验证，例如在末尾添加调用函数和类：

```
print(count(6, 6))
c = Count(8, 8)
print(c.count())
```

运行 mymodule.py 文件，运行结果：

```
6 与 6 的乘积是: 36
8 与 8 的乘积是: 64
```

可以正常工作，将添加这块代码再去掉。

在 test.py 文件中导入该模块，模块名即文件名：

```
import mymodule

print(mymodule.count(6, 6))
c = mymodule.Count(8, 8)
print(c.count())
```

运行 test.py 文件，运行结果一样导入正常。

有时模块名很长，为方便书写，在导入时候可以起个别名，例如：

```
import mymodule as mm
print(mm.count(6, 6))
```

除了直接 import 模块名称外，也可以直接导入只导入该模块的部分功能，例如：

```
from mymodule import count

print(count(6, 6))
c = Count(8, 8)
print(c.count())
```

运行结果：

```
6 与 6 的乘积是: 36
Traceback (most recent call last):
  File "C:/Users/zhenl/Desktop/test.py", line 898, in <module>
    c = Count(8, 8)
NameError: name 'Count' is not defined
```

提示没定义，这是因为只导入了该模块中的 count() 函数，其余的并没有导入到当前程序中，这样好处是：

- 当模块中代码量较大时，导入全部会产生不必要的消耗，一般只对要用的功能导入
- 引用时可以不加模块名，显得更为简洁

10.2 `__name__ == '__main__'` 作用

`mymodule.py` 作为一个模块，我们希望保留末尾测试代码，即上面调用函数和类，但也不希望再导入模块的时候执行。该怎么办呢？

我们可以利用 Python 文件都有一个内置属性 `__name__` 实现，如果直接运行 Python 文件，`__name__` 的值是 `"__main__"`，如果 `import` 一个模块，那么模块的 `__name__` 的值是“文件名”。

在 `mymodule.py` 文件最后增加打印 `__name__` 内置属性：

```
print(__name__)
# 运行结果
__main__
```

与预期一样，打印出了 `"__main__"`，再在 `test.py` 中导入这个模块：

```
import mymodule
```

运行 `test.py` 文件，运行结果：

```
mymodule
```

打印出了模块名，这个结果输出就是 `mymodule.py` 中的 `print(__name__)`。

所以，我们就可以在 `mymodule.py` 里面判断 `__name__` 值：

```
#!/usr/bin/python
def count(a, b):
    result = a * b
    return f"{a} 与 {b} 的乘积是: {result}"

class Count():
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def count(self):
        result = self.a * self.b
        return f"{self.a} 与 {self.b} 的乘积是: {result}"

if __name__ == "__main__":
    print("我在手动执行这个程序")
    print(count(6, 6))
    c = Count(8, 8)
    print(c.count())
```

运行 `mymodule.py` 文件，会正常执行最后这段测试代码。

再次运行 `test.py` 文件，会发现已不再打印。

10.3 模块帮助文档

我们知道，在定义函数或者类时，可以为其添加说明文档，以方便用户清楚的知道该函数或者类的功能。自定义模块也不例外，也可以添加说明文档，与函数或类的添加方法相同，即只需在模块开头的位置定义一个字符串即可：

```
#!/usr/bin/python
"""
count()函数用于计算两个数值乘积
Count 类也是计算两个数值乘积，其中包含 count() 方法
"""

def count(a, b):
    result = a * b
    return f"{a} 与 {b} 的乘积是: {result}"
```

然后通过 `__doc__` 内置属性查看了：

```
import mymodule
print(mymodule.__doc__)
```

也可以使用 `help()` 函数查看：

```
import mymodule
help(mymodule)
```

结果更详细，包括这个模块的类方法、属性等信息。

10.4 导入模块新手容易出现的问题

有一个新手经常犯的错误，在写一个模块时候，往往习惯性把文件名写成模块名，例如需要导入 `os` 系统模块，文件名就写 `os.py`，这就会引发下面错误：

```
import os
print(os.name)
```

运行 `os.py` 文件，运行结果：

```
Fatal Python error: initsite: Failed to import the site module
Traceback (most recent call last):
  File "D:\Python3.7\lib\site.py", line 73, in <module>
    import os
  File "C:\Users\zhenl\Desktop\devops\os.py", line 2, in <module>
    print(os.name)
AttributeError: module 'os' has no attribute 'name'
```

抛出异常，这是因为文件名即模块名，`import` 默认会从当前路径查找，就会导致 Python 解释器认为自己就是要导入的模块。所以文件名不要与引用的模块名相同。

第十一章 Python 常用标准库使用

什么是标准库？

在 Python 发行版中包含一系列非常有用的模块，这些模块也称为标准库；Python 模块是一个 Python 文件，以 `.py` 结尾，包含了 Python 对象定义和语句。模块让你能够有逻辑地组织代码，让你的代码更易懂、可复用。

标准库如何使用？

这些模块可以立刻使用，无需额外安装。

Python 的标准库应用非常广泛，提供了一系列功能帮助解决日常编程，非常有必要学习下，为后面应用开发打好基础。

官方文档：<https://docs.python.org/zh-cn/3.8/library/index.html>

11.1 os（操作系统管理）

os 模块主要对目录或文件操作。

os 模块以下常用方法：

方法	描述	示例
os.name	返回操作系统类型	返回值"posix"代表 linux，"nt"代表 windows。sys.platform 有更详细的描述。
os.environ	以字典形式返回系统变量	返回家目录： >>> os.environ['HOME']
os.putenv(key, value)	改变或添加环境变量	直接调用 putenv() 设置环境变量会影响由 os.system(), popen(), fork() 和 execv() 发起的子进程。不会影响 os.environ，所以推荐 os.environ 直接赋值，例如 os.environ["NAME"]="aliang"
os.listdir(path='.')	列表形式列出目录下所有目录和文件名	>>> os.listdir("/opt/")
os.getcwd()	获取当前路径	>>> os.getcwd() ' /root'
os.chdir(path)	改变当前工作目录到指定目录	>>> os.chdir('/opt') >>> os.getcwd() ' /opt'
os.mkdir(path [, mode=0777])	创建目录	>>> os.mkdir('/tmp/est')
os.makedirs(path [, mode=0777])	递归创建目录	>>> os.makedirs('/tmp/abc/abc')
os.rmdir(path)	移除空目录	>>> os.makedirs('/tmp/abc/abc')
os.remove(path)	移除文件	>>> os.remove("/opt/error.log")

<code>os.rename(old, new)</code>	重命名文件或目录	<pre>>>> os.rename("/opt/a", "/opt/a2")</pre>
<code>os.stat(path)</code>	获取文件或目录属性	<pre>>>> os.stat("/opt/a.log")</pre>
<code>os.chown(path, uid, gid)</code>	改变文件或目录所有者	<pre>>>> os.chown("/opt/a.log", 0, 0)</pre>
<code>os.chmod(path, mode)</code>	改变文件访问权限	<pre>>>> os.chmod('/opt/a.tar.gz', 0777)</pre>
<code>os.symlink(src, dst)</code>	创建软链接	<pre>>>> os.symlink("/opt/a.log", "/opt/b.log")</pre>
<code>os.unlink(path)</code>	移除软链接	<pre>>>> os.unlink('/opt/b.log')</pre>
<code>os.urandom(n)</code>	返回随机字节，适合加密使用	<pre>>>> os.urandom(2) '\x88\x19'</pre>
<code>os.getuid()</code>	返回当前进程 UID	
<code>os.getlogin()</code>	返回登录用户名	
<code>os.getpid()</code>	返回当前进程 ID	
<code>os.kill(pid, sig)</code>	发送一个信号给进程	
<code>os.walk(path)</code>	目录树生成器，生成一个三元组 (dirpath, dirnames, file names)	<pre>for root, dir, file in os.walk("/opt"): print(f"目录: {root}, 目录名: {dir}, 文件名: {file}")</pre>
<code>os.system(command)</code>	执行 shell 命令，不能存储结果	<pre>>>> os.system('ls')</pre>
<code>os.popen(command [, mode='r' [, bufsize]])</code>	打开管道来自 shell 命令，并返回一个文件对象	<pre>>>> result = os.popen('ls') >>> result.read()</pre>

其中包含的 `os.path` 类用于获取文件属性。

方法	描述	示例
<code>os.path.basename(path)</code>	返回最后一个文件或目录名	<pre>>>> os.path.basename('/home/user/a.sh') 'a.sh'</pre>
<code>os.path.dirname(path)</code>	返回最后一个文件所属目录	<pre>>>> os.path.dirname('/home/user/a.sh') '/home/user'</pre>

<code>os.path.abspath(path)</code>	返回一个绝对路径	<pre>>>> os.path.abspath('a.sh') '/home/user/a.sh'</pre>
<code>os.path.exists(path)</code>	判断路径是否存在，返回布尔值	<pre>>>> os.path.exists('/home/user/abc') True</pre>
<code>os.path.isdir(path)</code>	判断是否是目录	
<code>os.path.isfile(path)</code>	判断是否是文件	
<code>os.path.islink(path)</code>	判断是否是链接	
<code>os.path.ismount(path)</code>	判断是否挂载	
<code>os.path.getatime(filename)</code>	返回文件访问时间戳	<pre>>>> os.path.getctime('a.sh') 1475240301.9892483</pre>
<code>os.path.getctime(filename)</code>	返回文件变化时间戳	
<code>os.path.getmtime(filename)</code>	返回文件修改时间戳	
<code>os.path.getsize(filename)</code>	返回文件大小，单位字节	
<code>os.path.join(a, *p)</code>	加入两个或两个以上路径，以正斜杠"/"分隔。常用于拼接路径，例如	<pre>>>> os.path.join('/home/user', 'test.py', 'a.py') '/home/user/test.py/a.py'</pre>
<code>os.path.split()</code>	分隔路径名	<pre>>>> os.path.split('/home/user/test.py') ('/home/user', 'test.py')</pre>
<code>os.path.splitext()</code>	分隔扩展名	<pre>>>> os.path.splitext('/home/user/test.py') ('/home/user/test', '.py')</pre>

11.2 sys（解释器交互）

`sys` 模块用于与 Python 解释器交互。

常用方法：

方法	描述	示例
<code>sys.argv</code>	从程序外部传递参数 <code>argv[0]</code> #代表本身名字 <code>argv[1]</code> #第一个参数	<pre>import sys arg_list = sys.argv arg_length = len(arg_list)</pre>

	<p>argv[2] #第二个参数 argv[3] #第三个参数 argv[N] #第 N 个参数 argv #参数以空格分隔存储到列表。</p>	<pre>for i in arg_list: if i == sys.argv[0]: print("脚本名称: %s" %i) continue print("参数为: %s" %i) print(" 参 数 长 度: %s" %(arg_length - 1))</pre> <p>以下是代码运行结果: [root@localhost ~]# python3.6 sys1.py a b c 脚本名称: test.py 参数为: a 参数为: b 参数为: c 参数长度: 3</p> <p>说明: argv 方法返回的是一个列表, 使用 for 循环遍历出元素, 也可以通过 len() 函数获取这个列表的长度从而知道输入的参数数量。需要注意的是: argv[0] 是脚本名, 实际长度是 len(sys.argv - 1)。</p>
sys.exit([status])	<p>sys.exit() 方法用于退出 Python 解释器。例如程序执行到某个位置, 不想继续执行就可以 sys.exit() 退出了。</p> <p>可选参数 arg 是一个给出退出状态的整数 (默认为 0) 或其他类型的对象。如果是一个整数, 0 被认为是“成功终止”, 非 0 被 Shell 认为是“异常终止”。大多数系统要求在 0-127 范围内, 否则可能会产生未定义的结果。像 Unix、Linux 系统特定退出状态码有特定含义的惯例, 例如 2 是命令行语法错误, 1 是其他类型错误。</p>	<pre>import sys print("Hello world!1") sys.exit() print("Hello world!2") # 运行结果 Hello world!1</pre> <p>在 test.py 程序里打印了两个“Hello world!”, 但实际输出只有第一个, 因为第一个下面执行 sys.exit() 退出 Python, 所以第二个并没有执行。通过 Shell 的 \$? 指令的获取到执行 test.py 的命令返回状态码 “0”。</p>
sys.path	<p>获取指定模块搜索路径的字符串集合, 如果自己写的模块在非默认搜索路径下, 需要指定模块搜索路径, 就可以 import 正确找到了。</p>	<pre>print(sys.path) # 运行结果 ['/root', '/usr/local/lib/python36.zip', '/usr/local/lib/python3.6', '/usr/local/lib/python3.6/lib- dynload',</pre>

		<pre> '/usr/local/lib/python3.6/site-packages'] </pre> <p>输出的是一个列表,里面包含了当前 Python 解释器所能找到的模块目录。</p> <p>如果想指定自己的模块目录,可以直接追加:</p> <pre> sys.path.append('/opt/scripts') print(sys.path) # 运行结果 ['/root', '/usr/local/lib/python3.6.zip', '/usr/local/lib/python3.6', '/usr/local/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/site-packages', '/opt/scripts'] </pre>
<code>sys.getdefaultencoding()</code>	获取系统当前编码	
<code>sys.platform</code>	获取当前系统平台 Linux = linux Windows = win32 Mac OS X = darwin	<pre> >>> sys.platform 'linux' </pre>
<code>sys.version</code>	获取 Python 版本	<pre> >>> sys.version '3.8.5 (default, Oct 14 2019, 23:29:39) \n[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)]' </pre>
<code>sys.stdin</code> <code>sys.stdout</code> <code>sys.stderr</code>	标准输入 标准输出 错误输出	

`sys.stdin, sys.stdout, sys.stderr`

- `sys.stdin` 用于所有交互式输入, 例如 `input()`
- `sys.stdout` 用于输出 `print()`、表达式语句以及 `input()` 提示
- `sys.stderr` 解释器自己的提示和错误消息被发送到 `stderr`

示例: `sys.stdin`

```

import sys
name = input("请输入你的名字: ")
print(name)
# 运行结果
请输入你的名字: aliang
aliang

```

这个例子是通过之前学到的 `input()` 接收一个标准输入, 返回一个 `string` 类型。

再试试 `sys.stdin` 获取用户的输入:

```

import sys

```

```
print("请输入你的名字：")
name = sys.stdin.readline()
print(name)
# 运行结果
请输入你的名字：
aliang
aliang
```

使用 `sys.stdin.readline()` 方法读取用户输入一行内容。

第一个“aliang”是输入的内容，第二个是 `print()` 输出的，与上面例子一样获取到用户输入。这就是 `sys.stdin` 的作用，获取交互式输入。

示例：sys.stdout

```
import sys
print("Hello world!")
# 运行结果
Hello world!
```

很简单的打印一个字符串就是标准输出。

再试试 sys.stdout 输出：

```
import sys
sys.stdout.write("Hello world!\n")
# 运行结果
Hello world!
```

同样输出，`print()` 函数会默认输出一个“\n”回车换行符，`sys.stdout.write()` 方法是将字符串写到 `stdout`，与也就是标准输出。

示例：sys.stderr

```
import sys
sys.stderr.write("错误消息\n")
# 运行结果
错误消息
```

这个输出与上面例子 `sys.stdout` 效果是一样的，没错，因为错误也是输出。

为了区分标准输出和错误，使用 Linux Shell 的重定向测试。

在 Unix、Linux 下，可以用 3 个数字（文件描述符）表示：0 标准输入，1 标准输出，2 标准错误。

```
[root@localhost ~]# python test.py 1>out.log
[root@localhost ~]# cat out.log
Hello world!
[root@localhost ~]# python test.py 2>out.log
Hello world!
[root@localhost ~]# cat out.log
```

第一次执行只重定向 1（标准输出）到 `out.log` 文件，结果写到文件中。

第二次执行只重定向 2（标准错误）到 `out.log` 文件，结果打印到控制台，文件为空。

结论：输出的为标准输出，标准错误不会处理。

```
[root@localhost ~]# python test.py 1>err.log
错误消息
```

```
[root@localhost ~]# cat err.log
[root@localhost ~]# python test.py 2>err.log
[root@localhost ~]# cat err.log
错误消息
```

第一次执行只重定向 1（标准输出）到 err.log 文件，结果打印到控制台，文件为空。

第二次执行只重定向 2（标准错误）到 err.log 文件，结果写到文件中。

结论：输出的为标准错误，标准输出不会处理。

11.3 platform（获取操作系统信息）

platform 模块用于获取操作系统详细信息。

常用方法：

方法	描述	示例
platform.platform()	返回操作系统平台	<pre>>>> platform.platform() Linux-3.10.0- 957.21.3.el7.x86_64-x86_64- with-centos-7.6.1810-Core</pre>
platform.uname()	返回操作系统信息	<pre>>>> platform.uname() (uname_result(system='Linux', node='k8s-master', release='3.10.0- 957.21.3.el7.x86_64', version='#1 SMP Tue Jun 18 16:35:19 UTC 2019', machine='x86_64', processor='x86_64'))</pre>
platform.system()	返回操作系统平台	<pre>>>> platform.system() 'Linux'</pre>
platform.version()	返回操作系统版本	<pre>>>> platform.version() '#1 SMP Tue Jun 18 16:35:19 UTC 2019'</pre>
platform.machine()	返回计算机类型	<pre>>>> platform.machine() 'x86_64'</pre>
platform.processor()	返回计算机处理器类型	<pre>>>> platform.processor() 'x86_64'</pre>
platform.node()	返回计算机网络名	<pre>>>> platform.node() 'localhost'</pre>
platform.python_version()	返回 Python 版本号	<pre>>>> platform.python_version()</pre>

		' 3.8.5'
--	--	----------

11.4 glob（查找文件）

glob 模块用于文件查找，支持通配符（*、?、[]）。

示例 1：查找目录中所有以.sh 为后缀的文件：

```
>>> glob.glob('/home/user/*.sh')
['/home/user/1.sh', '/home/user/b.sh', '/home/user/a.sh', '/home/user/sum.sh']
```

示例 2：查找目录中出现单个字符并以.sh 为后缀的文件：

```
>>> glob.glob('/home/user/?.sh')
['/home/user/1.sh', '/home/user/b.sh', '/home/user/a.sh']
```

示例 3：查找目录中出现 a.sh 或 b.sh 的文件：

```
>>> glob.glob('/home/user/[a|b].sh')
['/home/user/b.sh', '/home/user/a.sh']
```

11.5 fileinput（处理多文件）

fileinput 模块用于遍历文件，适合处理多文件。

常用方法：

方法	描述
fileinput.input(files=None, inplace=False, backup='', *, mode='r', openhook=None)	files: 文件路径，多文件这样写['1.txt','2.txt'] inplace: 是否将标准输出写到原文件，默认是 0，不写 backup: 备份文件扩展名，比如.bak mode: 读写模式，默认 r，只读
fileinput.isfirstline()	检查当前行是否是文件的第一行
fileinput.lineno()	返回当前已经读取行的数量
fileinput.fileno()	返回当前文件数量
fileinput.filelineno()	返回当前读取行的行号
fileinput.filename()	返回当前文件名

示例内容：

```
# a.txt
a
b
c
# b.txt
```

x
y
z

遍历多文件：

```
import fileinput
for line in fileinput.input(files=['a.txt', 'b.txt']):
    print("当前正在操作文件: %s" %fileinput.filename())
    print("文件行号: %s" %fileinput.filelineno())
    print("当前行内容: %s" %line)
```

运行结果：

当前正在操作文件: a.txt
文件行号: 1
当前行内容: a

当前正在操作文件: a.txt
文件行号: 2
当前行内容: b

当前正在操作文件: a.txt
文件行号: 3
当前行内容: c
当前正在操作文件: b.txt
文件行号: 1
当前行内容: x

当前正在操作文件: b.txt
文件行号: 2
当前行内容: y

当前正在操作文件: b.txt
文件行号: 3
当前行内容: z

11.6 shutil（文件管理）

shutil 模块用于文件或目录拷贝，归档。

常用方法：

方法	描述
<code>shutil.copyfile(src, dst)</code>	复制文件
<code>shutil.copytree(src, dst)</code>	复制文件或目录
<code>shutil.move(src, dst)</code>	移动文件或目录

<code>shutil.rmtree(path, ignore_errors=False, onerror=None)</code>	递归删除目录。需注意 <code>os.rmdir()</code> 不能删除有文件的目录
<code>shutil.make_archive(base_name, format, root_dir=None, base_dir=None, verbose=0, dry_run=0, owner=None, group=None, logger=None)</code>	创建 zip 或 tar 归档文件。 base_name: 要创建归档文件名 format: 归档文件格式, 有 zip、tar、bztar、gztar root_dir: 要压缩的目录 base_dir: ? 用法: <code>shutil.make_archive('wp', 'zip', '/root/wordpress')</code>

小结:

如果你想操作文件路径, 请参考 `os.path` 模块

如果你想读写一个文件请参考第五章 `open()`

如果你想读取通过命令行传入的所有文件中的所有行, 请参考 `fileinput` 模块

对于高级文件和目录处理, 请参考 `shutil` 模块。

11.7 math (数字)

`math` 模块用于数字处理。

常用的方法:

方法	描述	示例
<code>math.pi</code>	返回圆周率	<pre>>>> math.pi 3.141592653589793</pre>
<code>math.ceil(x)</code>	返回 x 浮动的上限	<pre>>>> math.ceil(5.2) 6.0</pre>
<code>math.floor(x)</code>	返回 x 浮动的下限	<pre>>>> math.floor(5.2) 5.0</pre>
<code>math.trunc(x)</code>	将数字截尾取整	<pre>>>> math.trunc(5.2) 5</pre>
<code>math.fabs(x)</code>	返回 x 的绝对值	<pre>>>> math.fabs(-5.2) 5.2</pre>
<code>math.fmod(x, y)</code>	返回 x%y(取余)	<pre>>>> math.fmod(5, 2) 1.0</pre>
<code>math.modf(x)</code>	返回 x 小数和整数	<pre>>>> math.modf(5.2) (0.200000000000000018, 5.0)</pre>

<code>math.factorial(x)</code>	返回 x 的阶乘	<pre>>>> math.factorial(5) 120</pre>
<code>math.pow(x, y)</code>	返回 x 的 y 次方	<pre>>>> math.pow(2, 3) 8.0</pre>
<code>math.sqrt(x)</code>	返回 x 的平方根	<pre>>>> math.sqrt(5) 2.2360679774997898</pre>

11.8 random（随机数）

`random` 模块用于生成随机数。

常用的方法：

方法	描述	示例
<code>random.randint(a, b)</code>	随机返回整数 a 和 b 范围内数字	<pre>>>> random.randint(1, 10) 4</pre>
<code>random.random()</code>	生成随机数，它在 0 和 1 范围内	<pre>>>> random.random() 0.7373251914304791</pre>
<code>random.randrange(start, stop[, step])</code>	返回整数范围的随机数，并可以设置只返回跳数	<pre>>>> random.randrange(1, 10, 2) 5</pre>
<code>random.sample(array, x)</code>	从数组中返回随机 x 个元素	<pre>>>> random.sample([1, 2, 3, 4, 5], 2) [2, 4]</pre>
<code>choice(seq)</code>	从序列中返回一个元素	<pre>>>> random.choice([1, 2, 3, 4, 5]) 3</pre>

11.9 subprocess（执行 Shell 命令）

`subprocess` 模块主要用于执行 Shell 命令，工作时会 fork 一个子进程去执行任务，连接到子进程的标准输入、输出、错误，并获得它们的返回代码。

这个模块将取代 `os.system`、`os.spawn*`、`os.popen*`、`popen2.*` 和 `commands.*`。

`subprocess` 的主要方法：

`subprocess.run()`，`subprocess.Popen()`，`subprocess.call`

在 Python 3.5 之后版本中，新增 `subprocess.run()` 函数，官方建议使用此函数来使用 `subprocess` 模块的功能。当然，也可以依旧可以直接使用底层 `Popen` 接口。

语法：`subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal_newlines=None, **other_popen_kwargs)`

常用参数：

参数	说明
args	要执行的 shell 命令，默认是一个字符串序列，如['ls', '-al']或('ls', '-al')；也可是一个字符串，如'ls -al'，同时需要设置 shell=True。
stdin stdout stderr	run() 函数默认不会捕获命令运行结果的正常输出和错误输出，可以设置 stdout=PIPE, stderr=PIPE 来从子进程中捕获相应的内容；也可以设置 stderr=STDOUT，使标准错误通过标准输出流输出。
shell	如果 shell 为 True，那么指定的命令将通过 shell 执行。
cwd	函数在执行子进程前改变当前工作目录为 cwd。
timeout	设置命令超时时间。如果命令执行时间超时，子进程将被杀死，并弹出 TimeoutExpired 异常。
check	如果 check 参数的值是 True，且执行命令的进程以非 0 状态码退出，则会抛出一个 CalledProcessError 的异常，且该异常对象会包含参数、退出状态码、以及 stdout 和 stderr (如果它们有被捕获的话)。
encoding	如果指定了该参数，则 stdin、stdout 和 stderr 可以接收字符串数据，并以该编码方式编码。否则只接收 bytes 类型的数据。

示例：

```
import subprocess
cmd = "pwd"
result = subprocess.run(cmd, shell=True, timeout=3, stderr=subprocess.PIPE, stdout=subprocess.PIPE)
print(result)
# 运行结果
CompletedProcess(args='pwd', returncode=0, stdout=b'/opt\n', stderr=b'')
```

run 方法返回 CompletedProcess 实例，可以直接从这个实例中获取命令运行结果：

```
print(result.returncode) # 获取命令执行返回状态码
print(result.stdout) # 命令执行标准输出
print(result.stderr) # 命令执行错误输出
# 运行结果
0
b'/opt\n'
b''
```

11.10 pickle（数据持久化）

pickle 模块实现了对一个 Python 对象结构的二进制序列化和反序列化。

主要用于将对象持久化到文件存储。

pickle 模块主要有两个函数：

- dump() 把对象保存到文件中（序列化），使用 load() 函数从文件中读取（反序列化）
- dumps() 把对象保存到内存中，使用 loads() 函数读取

示例 1：

```
import pickle
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
```

```
with open("data.pkl", "wb") as f:
    pickle.dump(computer, f)
```

运行 py 文件会在本地生成一个二进制文件 data.pkl，里面保存了 computer 字典内容。
使用 load() 从文件中读取数据，重构原理的 Python 对象：

```
import pickle
with open("data.pkl", "rb") as f:
    print(pickle.load(f))
# 运行结果
{'主机': 5000, '显示器': 1000, '鼠标': 60, '键盘': 150}
```

示例 2:

```
import pickle
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
obj = pickle.dumps(computer)
print(type(obj))
print(obj)
# 运行结果
<class 'bytes'>
b'\x80\x03}q\x00(X\x06\x00\x00\x00\xe4\xb8\xbb\xe6\x9c\xbaq\x01M\x88\x13X\t\x00\x00\x00\xe6\x98\xbe\xe7\xa4\xba\xe5\x99\xa8q\x02M\xe8\x03X\x06\x00\x00\x00\xe9\xbc\xa0\xe6\xa0\x87q\x03K<X\x06\x00\x00\x00\xe9\x94\xae\xe7\x9b\x98q\x04K\x96u.'
```

反序列化:

```
obj = pickle.loads(obj)
print(type(obj))
print(obj)
# 运行结果
<class 'dict'>
{'主机': 5000, '显示器': 1000, '鼠标': 60, '键盘': 150}
```

11.11 json (JSON 编码和解码)

JSON 是一种轻量级数据交换格式，一般 API 返回的数据大多是 JSON、XML，如果返回 JSON 的话，需将获取的数据转换成字典，方面在程序中处理。

json 与 pickle 有相似的接口，主要提供两种方法：

- dumps() 对数据进行编码
- loads() 对书籍进行解码

示例：将字典类型转换为 JSON 对象

```
import json
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
json_obj = json.dumps(computer)
print(type(json_obj))
print(json_obj)
# 运行结果
<class 'str'>
```

```
{ "\u4e3b\u673a": 5000, "\u663e\u793a\u5668": 1000, "\u9f20\u6807": 60, "\u952e\u76d8": 150 }
```

将 JSON 对象转换为字典：

```
import json
data = json.loads(json_obj)
print(type(data))
print(data["主机"])
```

如果你要处理的是文件而不是 Python 对象，可以使用 `json.dump()` 和 `json.load()` 来编码和解码 JSON 数据。例如：

```
import json
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
# 写入 JSON 数据
with open("data.json", "w") as f:
    json.dump(computer, f)

# 读取 JSON 数据
with open("data.json ") as f:
    data = json.load(f)
    print(data)
```

JSON 与 Python 解码后数据类型：

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

11.12 time（时间访问和转换）

`time` 模块用于满足简单的时间处理，例如获取当前时间戳、日期、时间、休眠。

方法	描述	示例
time.ctime(seconds)	返回当前时间时间戳	>>> time.ctime() 'Sat Sep 12 17:07:50 2020'
time.localtime([seconds])	当前时间，以 struct_time 时间类型返回	>>> time.localtime() time.struct_time(tm_year=2020, tm_mon=9, tm_mday=12, tm_hour=17, tm_min=8, tm_sec=3, tm_wday=5, tm_yday=256, tm_isdst=0)
time.mktime(tuple)	将一个 struct_time 时间类型转换成时间戳	>>> time.mktime(time.localtime()) 1599901698.0
time.sleep(seconds)	延迟执行给定的秒数	>>> time.sleep(1.5)
time.strftime(format[, tuple])	将元组时间转换成指定格式。 [tuple]不指定默认当前时间	>>> time.strftime('%Y-%m-%d %H:%M:%S') '2020-09-12 17:08:34' 时间戳格式化： >>> time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())) '2020-09-12 17:08:45'
time.time()	返回当前时间时间戳	>>> time.time() 1599901758.07539

time.strftime (format [, t])
根据 format 参数指定，将元组或 struct_time 时间转换为字符串。
示例：

```
>>> import time
>>> strftime("%Y-%m-%d %H:%M:%S")
'2018-06-18 10:40:17'
```

如果不指定 t 可选参数，默认是当前时间。

```
>>> strftime("%Y-%m-%d %H:%M:%S", time.localtime())
'2018-06-18 10:41:19'
```

format 符号表示如下：

符号	描述
%a	简化星期名称，如 Sat

%A	完整星期名称，如 Saturday
%b	简化月份名称，如 Nov
%B	完整月份名称，如 November
%c	当前时区日期和时间
%d	天
%H	24 小时制小时数（0-23）
%I	12 小时制小时数（01-12）
%j	365 天中第多少天
%m	月
%M	分钟
%p	AM 或 PM，AM 表示上午，PM 表示下午
%S	秒
%U	一年中第几个星期
%w	星期几
%W	一年中第几个星期
%x	本地日期，如 '11/12/16'
%X	本地时间，如 '17:46:20'
%y	简写年名称，如 16
%Y	完整年名称，如 2016
%Z	当前时区名称（PST：太平洋标准时间）
%%	代表一个%号本身

11.13 datetime（日期和时间）

datetime 模块用于处理更复杂的日期和时间。

datetime 库提供了以下几个类：

类	描述
<code>datetime.date</code>	日期，年月日组成
<code>datetime.datetime</code>	包括日期和时间
<code>datetime.time</code>	时间，时分秒及微秒组成
<code>datetime.timedelta</code>	时间间隔
<code>datetime.tzinfo</code>	时区信息对象

导入模块：

```
from datetime import date, datetime, time, timedelta
```

`datetime.date` 类有以下常用方法：

方法	描述	描述
<code>date.strftime()</code>	根据 <code>datetime</code> 自定义时间格式	<pre>>>> date.strftime(datetime.now(), ' %Y-%m-%d %H:%M:%S') ' 2020-09-12 17:14:45'</pre>
<code>date.today()</code>	返回当前系统日期	<pre>>>> date.today() datetime.date(2020, 9, 12)</pre>
<code>date.weekday()</code>	根据日期返回星期几，周一是 0，以此类推	<pre>>>> date.weekday(date.today()) 5</pre>
<code>date.fromtimestamp()</code>	根据时间戳返回日期对象	<pre>>>> date.fromtimestamp(time.time()) datetime.date(2020, 9, 12)</pre>
<code>date.isocalendar()</code>	根据日期返回日历（年，第几周，星期几）	<pre>>>> date.isocalendar(date.today()) (2020, 37, 6)</pre>
<code>date.isoformat()</code>	返回 ISO 8601 格式时间	<pre>>>> datetime.isoformat(datetime.now()) ' 2020-09-12T17:20:51.046736'</pre>

日常编程中经常定义时间，以下两个方法用的多写：

```
>>> from datetime import date
>>> date.strftime(datetime.now(), ' %Y-%m-%d %H:%M:%S' )
```



```
'2018-06-18 14:08:54'
>>> date.isoformat(date.today())
'2018-06-18'
```

datetime.datetime 类有以下常用方法：

方法	描述	示例
<code>datetime.now()</code> <code>datetime.today()</code>	获取当前系统时间对象	<pre>>>> datetime.now() datetime.datetime(2020, 9, 12, 17, 20, 42, 477148)</pre>
<code>datetime.date()</code>	返回时间日期对象，年月日	<pre>>>> datetime.date(datetime.now()) datetime.date(2020, 9, 12)</pre>
<code>datetime.time()</code>	返回时间对象，时分秒	<pre>>>> datetime.time(datetime.now()) datetime.time(17, 21, 29, 753630)</pre>
<code>datetime.utcnow()</code>	UTC 时间，比中国时间快 8 个小时	<pre>>>> datetime.utcnow() datetime.datetime(2020, 9, 12, 9, 21, 47, 372973)</pre>

datetime.time 类有以下常用方法：

方法	描述	示例
<code>time.max</code>	所能表示的最大时间	<pre>>>> time.max datetime.time(23, 59, 59, 999999)</pre>
<code>time.min</code>	所能表示的最小时间	<pre>>>> time.min datetime.time(0, 0)</pre>
<code>time.resolution</code>	时间最小单位，1 微妙	<pre>>>> time.resolution datetime.timedelta(0, 0, 1)</pre>

datetime.timedelta 类：

示例：获取昨天日期

```
from datetime import timedelta, date
yesterday = date.today() - timedelta(days=1)
yesterday = date.isoformat(yesterday)
print(yesterday)
```

示例 2：获取明天日期

```
tomorrow = date.today() + timedelta(days=1)
# 将时间对象格式化
tomorrow = date.isoformat(tomorrow)
print(tomorrow)
```

11.14 urllib (HTTP 访问)

urllib 模块用于访问 URL。urllib2 是 urllib 的增强版，新增了一些功能，比如 Request() 用来修改 Header 信息。但是 urllib2 还去掉了一些好用的方法，比如 urlencode() 编码序列中的两个元素（元组或字典）为 URL 查询字符串。

一般情况下这两个库结合着用，那我们也结合着了解下。

urllib 包含以下模块：

- urllib.request 打开和读取 URL
- urllib.error 包含 urllib.request 抛出的异常
- urllib.parse 用于解析 URL
- urllib.robotparser 用于解析 robots.txt 文件

用的最多是 urllib.request 模块，它定义了适用于在各种复杂情况下打开 URL（主要为 HTTP）的函数和类，例如基本认证、摘要认证、重定向、cookies 等。

urllib.request 常用方法：

方法	描述
urllib.request.urlopen (url, data = None, [timeout,] *, cafile = None, capath = None, cadefault = False, context = None)	读取指定 URL，创建类文件对象。 data 是随着 URL 提交的数据（POST）
urllib.request.build_opener([handler, ...])	构造 opener
urllib.request.install_opener(opener)	把新构造的 opener 安装到默认的 opener 中，以后 urlopen() 会自动调用

urlopen() 函数本身就是一个 opener，但不支持验证、cookie 或其他 HTTP 高级功能，要想使用更高级功能访问 URL，必须使用 build_opener() 函数来构建自定义 opener 对象。参数 handler 是处理程序对象的实例，这些处理程序目的是向得到的 opener 对象添加各种功能。默认情况下，始终使用以下处理程序创建 opener。

ProxyHandler	通过代理重定向请求
UnknownHandler	处理所有未知 URL 的处理程序
HTTPHandler	通过 HTTP 打开 URL
HTTPDefaultErrorHandler	通过引发 HTTPError 异常来处理 HTTP 错误
HTTPRedirectHandler	处理 HTTP 重定向
FTPHandler	通过 FTP 打开 URL
FileHandler	打开本地文件
HTTPSHandler	如果 Python 支持 SSL，这个也将被添加

urllib.request 常用类：

类	描述
<code>urllib.request.Request(url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None)</code>	一般访问 URL 用 <code>urllib.urlopen()</code> ，如果要修改 header 信息就会用到这个。 data 是随着 URL 提交的数据，将会把 HTTP 请求 GET 改为 POST。headers 是一个字典，包含提交头的键值对应内容。 参数说明： url: 一个有效的网址字符串 data: 发送服务器的数据 headers: 一个字典，提交到请求头的键值 origin_req_host: 请求的原始主机 unverifiable: 布尔值，表示请求是否不可验证 RFC 2965 method: HTTP 请求方法，默认是“GET”
<code>urllib.request.HTTPCookieProcessor(cookiejar=None)</code>	Cookie 处理器
<code>urllib.request.ProxyHandler(proxies=None)</code>	代理处理器
<code>urllib.request.HTTPBasicAuthHandler(password_mgr=None)</code>	HTTP 基础认证处理器
<code>urllib.request.HTTPHandler</code>	HTTP 处理器

urllib.parse 模块用于将 URL 字符串拆分或者组合。

方法	描述
<code>urllib.parse.urlencode(query, doseq = False, safe = '' , encoding = None, errors = None, quote_via = quote_plus)</code>	将序列中的两个元素（元组或字典）转换为 URL 查询字符串
<code>urllib.parse.quote(string, safe = '/' , encoding = None, errors = None)</code>	将字符串中的特殊符号转十六进制表示。 如： <code>quote('abc def')</code> -> <code>'abc%20def'</code>
<code>urllib.parse.unquote(string, encoding = 'utf-8', errors = 'replace')</code>	与 quote 相反

在工作中，经常有调用别人接口的需求，为让 HTTP 接口识别请求的地址，需要对 URL 进行编码，就会用到以下两个模块了。

(1) urllib.parse.quote

该函数是将 URL 中的一个字符串或者特殊符号转换为十六进制表示（URL 编码）。

```
urllib.parse.quote(string, safe='/', encoding=None, errors=None)
```

示例：

```
>>> import urllib.parse
>>> base_url = 'https://s.taobao.com/search?'
>>> url = base_url + 'q=' + urllib.parse.quote('手机')
>>> print(url)
https://s.taobao.com/search?q=%E6%89%8B%E6%9C%BA
```

当对 URL 编码后就可以使用 `urlopen()` 函数访问了。

(2) `urllib.parse.urlencode`

该函数是将字典或包含两个元素元组或序列转换为 URL 查询字符串。

```
urllib.parse.urlencode(query, doseq = False, safe = '', encoding = None, errors = None,
quote_via = quote_plus )
```

示例：

```
>>> import urllib.parse
>>> parameters = {'q': '手机'}
>>> url = base_url + urllib.parse.urlencode(parameters)
>>> print(url)
https://s.taobao.com/search?q=%E6%89%8B%E6%9C%BA
```

小结：

`quote()` 适于单个字符编码，`urlencode` 需要用键值形式数组，后者用的更多些。

如果需要对 URL 解码可以使用 `unquote()` 函数：

```
>>> print(urllib.parse.unquote(url))
https://s.taobao.com/search?q=手机
```

接下来结合示例学习 `urllib` 模块，先发起一个简单的 URL 请求：

```
from urllib import request
res = request.urlopen("http://www.ctnrs.com")
```

请求对象，返回一个 `HTTPResponse` 类型的对象，包含的方法和属性：

方法	描述	示例
<code>getcode()</code>	获取响应的 HTTP 状态码	<code>res.getcode()</code> 200
<code>geturl()</code>	返回真实 URL。有可能 URL3xx 跳转，那么这个将获得跳转后的 URL	<code>res.geturl()</code> 'http://www.ctnrs.com'
<code>headers</code>	返回服务器 header 信息	<code>print(res.headers)</code> <code>res.headers['Date']</code>
<code>read(size=-1)</code>	返回网页所有内容。size 正整数指定读取多少字节	

<code>readline(limit=-1)</code>	读取下一行。size 正整数指定读取多少字节	<code>res.readline()</code> <code>b'<html lang="en-us">\n'</code>
<code>readlines(hint=0, /)</code>	列表形式返回网页所有内容，以列表形式返回。sizehint 正整数指定读取多少字节	

示例：

1、伪装 chrome 浏览器访问

应用场景：有些网页为了防止别人恶意采集其信息所以进行了一些反爬虫的设置，例如限制未知的 UserAgent 访问，因此一般爬虫程序设置 UserAgent，模拟成浏览器去访问。

直接向 URL 发起请求：

```
from urllib import request
res = request.urlopen("http://www.ctnrs.com")
```

在服务器查看 Nginx 访问日志：

```
152.136.157.95 - - [12/Sep/2020:20:21:00 +0800] "GET / HTTP/1.1" 200 20664 "-" "Python-urllib/3.6"
```

日志中"Python-urllib/3.6"则是用户代理信息。

```
from urllib import request
url = "http://www.ctnrs.com"
user_agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36"
header = {"User-Agent": user_agent}
req = request.Request(url, headers=header)
# req.add_header("User-Agent", user_agent) # 或者这种方式添加 UA
res = request.urlopen(req)
print(res.getcode())
```

再查看日志：

```
27.189.225.231 - - [12/Sep/2020:20:19:19 +0800] "GET / HTTP/1.1" 200 20664 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36"
```

已成功模拟我们定义的 UA。

2、提交用户表单

应用场景：爬取的网页需要登录后才能看到，需要程序模拟登录。

打开目标网站登录页面，从浏览器开发工具中分析提交用户名和密码的表单 id，放到 post_data 里面。

```
url = "http://home.51cto.com/index"
post_data = {"loginform-username": "user1", "loginform-password": "123456"}
post_data = parse.urlencode(post_data).encode("utf8") # 必须转为 bytes 类型。字典转换为 URL 接受的编码格式 loginform-username=test&loginform-password=123456
req = request.Request(url, data=post_data, headers=header)
res = request.urlopen(req)
print(res.read()) # 登录后网页内容
```

3、保存 cookie 到变量中（HTTPCookieProcessor）

什么是 Cookie？

某些网站为了辨别用户身份、进行 session 跟踪而存储在用户本地终端上的数据。例如用户浏览器登录到一个网站，用户只要在当前浏览器访问，在 session 一定有效时间内访问，始终是登录状态，如果用户换了浏览器或者清理了浏览器 cookie，那么会丢失登录状态。为避免程序每次请求 URL 都要登录，通过保存 cookie 来实现。

urllib 不支持保存 cookie 功能，所以需要使用 http.cookiejar 模块来存储 HTTP cookies。

在 Python2.x 版本中 http.cookiejar 模块名称是 cookielib，CookieJar 类是用来捕获 cookie 的。

```
from urllib import request
from http import cookiejar
url = "https://www.baidu.com"
cookie = cookiejar.CookieJar() # 实例化 CookieJar 对象来保存 cookie
handler = request.HTTPCookieProcessor(cookie) # 创建 cookie 处理器
opener = request.build_opener(handler) # 通过 handler 构造 opener
res = opener.open(url)
for item in cookie:
    print(item.name, item.value)
# 运行结果
BAIDUID ACF64042CEE1EA29959161E0F42CF1D3:FG=1
BIDUPSID ACF64042CEE1EA29DC8DCC32071BB0D9
PSTM 1599916349
BD_NOT_HTTPS 1
```

urlopen() 本身就是一个 opener，但本身不支持对 Cookie 处理，所有需要新构造一个 opener。

也可以将 cookie 保存到文件中：

1) 保存 cookie 到文件

```
import urllib.request
import http.cookiejar

cookie_file = 'cookie.txt'
# 保存 cookie 到文件
cookie = http.cookiejar.MozillaCookieJar(cookie_file)
# 创建 cookie 处理器
cookie_handler = urllib.request.HTTPCookieProcessor(cookie)
# 通过 handler 构造 opener
opener = urllib.request.build_opener(cookie_handler)
response = opener.open("http://www.baidu.com")
# 保存
cookie.save()
```

2) 从文件中读取 cookie

```
import urllib.request
import http.cookiejar

# 实例化对象
cookie = http.cookiejar.MozillaCookieJar()
```

```
# 从文件中读取 cookie
cookie.load("cookie.txt")
# 创建 cookie 处理器
cookie_handler = urllib.request.HTTPCookieProcessor(cookie)
# 通过 handler 构造 opener
opener = urllib.request.build_opener(cookie_handler)
response = opener.open("http://www.baidu.com")
```

4、使用代理服务器访问 URL (ProxyHandler)

应用场景：使用同一个 IP 去爬取同一个网站上的网页，久了之后会被该网站服务器屏蔽。

解决方法：使用代理服务器，通过代理服务器去访问网页，服务器上看到的不是我们真实 IP，而是代理服务器 IP。

```
from urllib import request
url = "http://www.ctnrs.com"

proxy_address = {"http": "http://117.69.178.36:9999"}
handler = request.ProxyHandler(proxy_address)
opener = request.build_opener(handler)
res = opener.open(url)
print(res.getcode())
```

默认 ProxyHandler 处理器不支持代理认证，这里使用 ProxyBasicAuthHandler 添加代理认证支持。

```
import urllib.request
# 创建一个支持基本 HTTP 认证的 OpenerDirector
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# 这一次我们没有安装 OpenerDirector，而是直接使用它
opener.open('http://www.example.com/login.html')
```

5、URL 访问认证 (HTTPBasicAuthHandler)

```
url = "http://www.ctnrs.com/test.html"
auth_handler = request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='None',
                          uri='http://www.ctnrs.com',
                          user='user1',
                          passwd='123456')
opener = request.build_opener(auth_handler)
res = opener.open(url)
```

6、下载文件

```
file_name = "nginx-1.18.0.tar.gz"
url = "http://nginx.org/download/" + file_name
f = request.urlopen(url).read()
```

```
with open(file_name, "wb") as data:
    data.write(f)
```

11.15 configparser（文件格式）

configparser 模块用于配置文件解析。
主要用到 configparser.ConfigParser() 类，对 ini 格式文件增删改查。
ini 文件固定结构：有多个部分块组成，每个部分有一个[标识]，并有多 key，每个 key 对应每个值，以等号“=”分隔。例如：

```
[Section1 Name]
Keyname1=value1
Keyname2=value2
... ..
[Section2 Name]
Keyname21=value21
Keyname22=value22
```

值的类型有三种：字符串、整数和布尔值。其中字符串可以不用双引号，布尔值为真用 1 表示，布尔值为假用 0 表示。注释以分号“;”开头。
configparser.ConfigParser() 类常用方法：

方法	描述
add_section(section)	创建一个新的部分配置
get(section, option, raw=False, vars=None)	获取部分中的选项值，返回字符串
getboolean(section, option)	获取部分中的选项值，返回布尔值
getfloat(section, option)	获取部分中的选项值，返回浮点数
getint(section, option)	获取部分中的选项值，返回整数
has_option(section, option)	检查部分中是否存在这个选项
has_section(section)	检查部分是否在配置文件中
items(section, raw=False, vars=None)	列表元组形式返回部分中的每一个选项
options(section)	列表形式返回指定部分选项名称
read(filenames)	读取 ini 格式的文件
remove_option(section, option)	移除部分中的选项
remove_section(section, option)	移除部分

sections()	列表形式返回所有部分名称
set(section, option, value)	设置选项值，存在则更新，否则添加
write(fp)	写一个 ini 格式的配置文件

举例说明，写一个 ini 格式文件，对其操作：

```
# config.ini
[host1]
host = 192.168.1.1
port = 22
user = root
pass = 123
[host2]
host = 192.168.1.2
port = 22
user = root
pass = 456
[host3]
host = 192.168.1.3
port = 22
user = aliang
pass = 789
```

1、获取部分中的键值

```
from configparser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
section = conf.sections()[0] # 获取随机的第一个部分标识
options = conf.options(section) # 获取部分中的所有键
key1 = options[0] # 获取第一个键
value1 = conf.get(section, key1) # 获取第一个键的值
key2 = options[1] # 获取第一个键
value2 = conf.get(section, key2) # 获取第一个键的值
print("%s: %s" % (key1, value1))
print("%s: %s" % (key2, value2))
# 运行结果
host: 192.168.1.1
port: 22
```

这里有意打出来了值的类型，来说明下 get() 方法获取的值都是字符串，如果有需要，可以 getint() 获取整数。测试发现，ConfigParser 是从下向上读取的文件内容！

2、遍历文件中的每个部分的每个字段

```
from configparser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
sections = conf.sections() # 获取部分名称 ['host3', 'host2', 'host1']
for section in sections:
```

```
options = conf.options(section) # 获取部分名称中的键 ['user', 'host', 'port', 'pass',
']
for option in options:
    value = conf.get(section, option) # 获取部分中的键值
    print(option + ": " + value)
    print("-----")
# 运行结果
host: 192.168.1.1
port: 22
user: root
pass: 123
-----
host: 192.168.1.2
port: 22
user: root
pass: 456
-----
host: 192.168.1.3
port: 22
user: aliang
pass: 789
-----
```

使用 items() 获取部分中的每个选项：

```
from configparser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
print(conf.items('host1'))
# 运行结果
[('host', '192.168.1.1'), ('port', '22'), ('user', 'root'), ('pass', '123')]
```

3、更新或添加选项

```
from configparser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
fp = open("config.ini", "w") # 写模式打开文件，供后面提交写的内容
conf.set("host1", "port", "2222") # 有这个选项就更新，否则添加
conf.write(fp) # 写入的操作必须执行这个方法
```

4、添加一部分，并添加选项

```
from configparser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
fp = open("config.ini", "w")
conf.add_section("host4") # 添加[host4]
conf.set("host4", "host", "192.168.1.4")
conf.set("host4", "port", "22")
conf.set("host4", "user", "user")
```

```
conf.set("host4", "pass", "666")
conf.write(fp)
```

5、删除一部分

```
from configparser import ConfigParser
conf = ConfigParser()
conf.read("config.ini")
fp = open("config.ini", "w")
conf.remove_section('host4') # 删除[host4]
conf.remove_option('host3', 'pass') # 删除[host3]的 pass 选项
conf.write(fp)
```

11.16 argparse（命令行参数解析）

argparse 模块用于命令行参数解析，能自动生成帮助文档。
学习这个工具目的是为后面写工具，给用户提供一个友好的使用方式。
从一个非常简单的例子开始：

```
[root@localhost ~]# cat argparse-1.py
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
以下是代码运行结果：
[root@localhost ~]# python3.6 argparse-1.py
[root@localhost ~]# python3.6 argparse-1.py --help
usage: argparse-1.py [-h]

optional arguments:
  -h, --help  show this help message and exit
[root@localhost ~]# python3.6 argparse-1.py --verbose
usage: argparse-1.py [-h]
argparse-1.py: error: unrecognized arguments: --verbose
[root@localhost ~]# python3.6 argparse-1.py foo
usage: argparse-1.py [-h]
argparse-1.py: error: unrecognized arguments: foo
```

从以上结果得知：

- 在没有任何选项的情况下运行脚本不会显示任何输出
- 在第二个执行开始可以看到 argparse 模块的用途了。几乎没做任何事情，但我们已经有一个很友好的帮助信息。
- 该 -help 选项也可以缩短 -h，这是唯一默认可接受的选项。

根据上面的例子，添加自定义的选项需要用到 `add_argument()` 函数。

```
ArgumentParser.add_argument(name or flags...[, action][, nargs][, const][, default]
[, type][, choices][, required][, help][, metavar][, dest])
```

参数说明：

name or flags: 名称或选项字符串列表, 例如 foo 或 -f, --foo

action: 参数存储的基本类型, 默认为 store

nargs: 命令行参数的数量

const: 保存一个常量

default: 默认值

type: 参数类型, 默认为 str

choices: 设置参数值的范围

required: 该选项是否必选, 默认 False

help: 参数简要描述

metavar: 帮助信息中显示的参数名称

dest: 要添加到返回对象的属性名称 parse_args()

示例:

```
[root@localhost ~]# cat argparse-1.py
#!/usr/local/bin/python3.6
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-H", "--hostname", help="hostname or ip", required=True)
parser.add_argument("-p", "--port", help="server port", required=True, type=int)
args = parser.parse_args()
print(args)
if args.hostname:
    print("主机名: %s" %args.hostname)
if args.port:
    print("端口: %s" %args.port)
```

以下是代码运行结果:

```
[root@localhost ~]# python3.6 argparse-1.py -H 192.168.1.10
usage: argparse-1.py [-h] -H HOSTNAME -p PORT
argparse-1.py: error: the following arguments are required: -p/--port
[root@localhost ~]# python3.6 argparse-1.py -H 192.168.1.10 -p abc
usage: argparse-1.py [-h] -H HOSTNAME -p PORT
argparse-1.py: error: argument -p/--port: invalid int value: 'abc'
[root@localhost ~]# python3.6 argparse-1.py -H 192.168.1.10 -p 22
Namespace(hostname='192.168.1.10', port=22)
主机名: 192.168.1.10
端口: 22
```

由于添加的两个选项都是必选的, 第一次执行没通过, 提示需要指定 -p/--port 选项, 第二次执行 -p 选项值错误, 因为设置的数据类型是数字, 第三次执行通过, 打印出 parser.parse_args() 存储传入的值, 默认是一个选项名对应一个值, 这样一来, 我们可以很轻易的拿到用户输入的值, 然后处理其他逻辑。

这个模块相比前面学到的 sys.argv 命令行参数要功能完善很多!

11.17 smtplib (发送邮件)

在写脚本时，放到后台运行，想知道执行情况，会通过邮件、短信、微信等方式通知管理员，邮件目前用的最多的通知方式。在 linux 下，Shell 脚本发送邮件告警是件很简单的事，有现成的邮件服务软件或者调用运营商邮箱服务器，对于 Python 来说，需要编写脚本调用邮件服务器来发送邮件。

Python 分别提供了收发邮件的库，smtplib、poplib 和 imaplib。

本章主要学习如何使用 smtplib 库发送各种形式的邮件内容。该库主要用 smtplib.SMTP() 类，使用这个类封装一个 SMTP 连接，语法如下：

```
smtplib.SMTP(host='', port=0, local_hostname=None, [ timeout, ] source_address= None )
```

SMTP 实例有以下常用方法：

方法	描述
SMTP.connect([host[, port]])	连接到指定的 SMTP 服务器
SMTP.login(user, password)	登录 SMTP 服务器
SMTP.sendmail(from_addr, to_addrs, msg, mail_options=[], rcpt_options=[])	发送邮件 from_addr: 邮件发件人 to_addrs: 邮件收件人 msg: 发送消息
SMTP.quit()	关闭 SMTP 会话
SMTP.close()	关闭 SMTP 服务器连接

为更好学习 smtplib 模块，我们下面写几个具体的示例来熟悉它的用法。

示例 1：发送文本邮件

先通过本地 SMTP 服务器发送邮件。

本地 SMTP 服务器，可以通过安装 sendmail、postfix 等服务提供。

例如安装 sendmail：

```
yum install sendmail -y
systemctl start sendmail
```

smtp 默认 25 端口，查看是否监听：

```
ss -anpt|grep 25
```

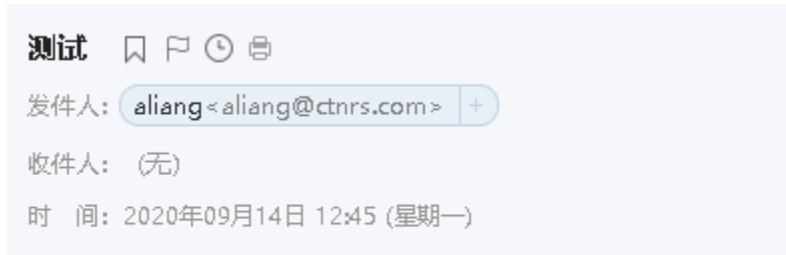
sendmail 服务运行，就可以使用了：

```
from smtplib import SMTP
s = SMTP("localhost")
to_mail = ["baojingtongzhi@163.com"] # 收件人列表，可以写多个
# 内容必须靠左，否则邮件头不识别
```

作者: 阿良

```
msg = '''\n
From: aliang@ctnrs.com\n
Subject: test\n
这是一份测试邮件'''.encode("utf8")\n
s.sendmail("aliang@ctnrs.com",to_mail,msg)\n
s.quit()
```

msg 对象里 From 表示发件人, Subject 是邮件标题, 换行后输入的是邮件内容。



这是一封测试邮件

上面示例使用本地 SMTP 服务器发送的邮件, 这种自建的 SMTP 服务器容易被当做垃圾邮件, 甚至大部分公有云也不允许自建 (禁止 25 端口), 所以一般情况下都采用个人邮箱或者企业邮箱发邮件。例如, 使用 163 邮箱发邮件:

```
from smtplib import SMTP\n
email_account = "baojingtongzhi@163.com"\n
account_password = "VCGXTBHXPVDVLGZ" # 授权密码\n
s = SMTP("smtp.163.com")\n
s.login(email_account, account_password)\n
to_mail = ["1121267855@qq.com"]\n
msg = f'''\\n\n
From: {email_account}\\n\n
Subject: 测试\\n\n
这是一封测试邮件'''.encode("utf8")\n
s.sendmail("baojingtongzhi@163.com",to_mail,msg)\n
s.quit()
```

现在一般个人邮箱需要授权码才可以在客户端收发邮件, 设置-> POP3/SMTP/IMAP->授权密码:

POP3/SMTP/IMAP

开启服务： IMAP/SMTP服务 已关闭 | [开启](#)
 POP3/SMTP服务 已开启 | [关闭](#)

POP3/SMTP/IMAP服务能让你在本地客户端上收发邮件，[了解更多](#) >

温馨提示：在第三方登录网易邮箱，可能存在邮件泄露风险，甚至危害Apple或其他平台账户安全

收取选项：
☒ 收取最近30天邮件
☐ 收取全部邮件

温馨提示：收取大量邮件，会耗费您更多的流量，建议您选择“收取最近30天邮件”

通知提醒：
☐ 开启客户端删除邮件提醒
 当邮件客户端大量删除邮件时，系统会发送提醒信息

授权密码管理： 授权码是用于登录第三方邮件客户端的专用密码。
 适用于登录以下服务：您开启的服务（例如POP3/IMAP/SMTP）、Exchange/CardDAV/CalDAV服务。

使用设备	启用时间	操作
设备1	2020.9.14	删除

[新增授权密码](#)

运行结果：

```
smtplib.SMTPDataError: (554, b'DT:SPM 163 smtp12,EMCowAC312M2915fNpBQAw--.41693S2 16000
58934,please see http://mail.163.com/help/help_spam_16.htm?ip=27.189.225.231&hostid=smt
p12&time=1600058934')
```

访问给出的163网址，SMTP554错误是：“554 DT:SUM 信封发件人和信头发件人不匹配”

在上面第一个示例中你会发现收件人是空的，看这样163的SMTP服务器拒绝的原因应该就是这样收件人没指定。

重新修改下msg对象，添加收件人：

```
msg = f'''
From: {email_account}
To: {','.join(to_mail)}
Subject: 测试
这是一封测试邮件'''.encode("utf8")
```

','.join(to_mail)转成字符串，以逗号分隔元素。


如果msg对象含带中文需要编码encode("utf8")

测试☆

发件人: **baojingtongzhi** <baojingtongzhi@163.com> (此地址未验证, 请意识识别) 

时间: 2020年9月14日 (星期一) 下午1:39

收件人: 可爱♪的你' <1121267855@qq.com>

发送者邮件地址未通过验证, 请勿轻信中奖、汇款等虚假信息, 勿轻易拨打陌生电话。  举报垃圾邮件

这是一封测试邮件

可以正常发送邮件了。msg 这个格式是 SMTP 规定的, 一定要遵守。

示例 2: 发送邮件并抄送

```
import smtplib
def sendMail(body):
    smtp_server = 'smtp.163.com'
    email_account = 'baojingtongzhi@163.com'
    account_password = 'VCGXTBHXPVDVLGZ'
    to_mail = ["1121267855@qq.com"]
    cc_mail = ["xxx@163.com"]
    from_name = 'monitor'
    subject = '监控'
    """

    msg = '''\
From: {email_account}
To: {'','.join(to_mail)}
Subject: 测试'''
    """

    # 或者写成列表再拼接, 相比上面顶头写更美化些
    mail = [
        "From: %s <%s>" % (from_name, account_password),
        "To: %s" % '','.join(to_mail),
        "Subject: %s" % subject,
        "Cc: %s" % '','.join(cc_mail),
        "",
        body
    ]
    msg = '\n'.join(mail).encode("utf8")

    try:
        s = smtplib.SMTP()
        s.connect(smtp_server, '25')
        s.login(email_account, account_password)
        s.sendmail(email_account, to_mail+cc_mail, msg)
        s.quit()
    except smtplib.SMTPException as e:
        print("Error: %s" %e)
```



```
if __name__ == "__main__":  
    sendMail("这是一封测试邮件")
```

监控 ☆


发件人: **monitor** <> (此地址未验证, 请注意识别) 

(由 baojingtongzhi@163.com 代发) 

时 间: 2020年9月14日 (星期一) 下午1:53

收件人: 可爱ノ的你' <1121267855@qq.com>

抄 送: zhenliang369 <zhenliang369@163.com>

发送者邮件地址未通过验证, 请勿轻信中奖、汇款等虚假信息, 勿轻易拨打陌生电话。  举报垃圾邮件

这是一封测试邮件

s.sendmail(from_mail, to_mail+cc_mail, msg) 在这里看到, 收件人和抄送人为什么放一起发送呢?

其实无论是收件人还是抄送人, 它们收到的邮件都是一样的, SMTP 都是认为收件人这样一封一封的发出。所以实际上并没有抄送这个概念, 只是在邮件头加了抄送人的信息罢了!

示例 3: 发送邮件带附件

由于 SMTP.sendmail() 方法不支持添加附件, 所以需要借助 email 模块来实现。email 模块是一个构造邮件和解析邮件的模块。

```
import smtplib  
from email.mime.text import MIMEText  
from email.mime.multipart import MIMEMultipart  
from email.header import Header  
from email import encoders  
from email.mime.base import MIMEBase  
from email.utils import parseaddr, formataddr  
  
# 格式化邮件地址  
def formatAddr(s):  
    name, addr = parseaddr(s)  
    return formataddr((Header(name, 'utf-8').encode(), addr))  
  
def sendMail(body, attachment):  
    smtp_server = 'smtp.163.com'  
    email_account = 'baojingtongzhi@163.com'  
    account_password = 'VCGXTBHXFPVDVLGZ'  
    to_mail = ["1121267855@qq.com"]  
  
    # 构造一个 MIMEMultipart 对象代表邮件本身  
    msg = MIMEMultipart()  
    # Header 对中文进行转码  
    msg['From'] = formatAddr('管理员 <%s>' % email_account)  
    msg['To'] = ', '.join(to_mail)  
    msg['Subject'] = Header('监控', 'utf-8')
```


```

# plain 代表纯文本
msg.attach(MIMEText(body, 'plain', 'utf-8'))

# 二进制方式模式文件
with open(attachment, 'rb') as f:
    # MIMEBase 表示附件的对象
    mime = MIMEBase('text', 'txt', filename=attachment) # 使用 MIMEBase 类构造附件并
添加到 msg 对象
    # filename 是显示附件名字
    mime.add_header('Content-Disposition', 'attachment', filename=attachment)
    # 获取附件内容
    mime.set_payload(f.read())
    encoders.encode_base64(mime)
    # 作为附件添加到邮件
    msg.attach(mime)
try:
    s = smtplib.SMTP()
    s.connect(smtp_server, "25")
    s.login(email_account, account_password)
    s.sendmail(email_account, to_mail, msg.as_string()) # as_string()把 MIMEText 对
象变成 str
    s.quit()
except smtplib.SMTPException as e:
    print("Error: %s" % e)
if __name__ == "__main__":
    sendMail('这是一封携带附件的测试邮件', 'test.txt')


```

监控 ☆

发件人: **baojingtongzhi** <baojingtongzhi@163.com> 

时 间: 2020年9月14日 (星期一) 下午2:06

收件人: 可爱ノ的你' <1121267855@qq.com>

附 件: 1 个 ( test.txt)

这是一封携带附件的测试邮件

附件(1 个)

普通附件 (✓ 已通过电脑管家云查杀引擎扫描)



test.txt (7 字节)

[预览](#) [下载](#) [收藏](#) [转存](#) ▼

示例 4：发送 HTML 邮件

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.header import Header
from email.utils import parseaddr, formataddr


# 格式化邮件地址
def formatAddr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))

def sendMail(body):
    smtp_server = 'smtp.163.com'
    email_account = 'baojingtongzhi@163.com'
    account_password = 'VCGXTBHXPVDVLGZ'
    to_mail = ["1121267855@qq.com"]

    # 构造一个 MIMEMultipart 对象代表邮件本身
    msg = MIMEMultipart()
    # Header 对中文进行转码
    msg['From'] = formatAddr('管理员 <%s>' % email_account)
    msg['To'] = ', '.join(to_mail)
    msg['Subject'] = Header('监控', 'utf-8')
    msg.attach(MIMEText(body, 'html', 'utf-8'))

    try:
        s = smtplib.SMTP()
        s.connect(smtp_server, "25")
        s.login(email_account, account_password)
        s.sendmail(email_account, to_mail, msg.as_string()) # as_string()把 MIMEText 对
象变成 str
        s.quit()
    except smtplib.SMTPException as e:
        print("Error: %s" % e)
if __name__ == "__main__":
    body = """
    <h1>测试邮件</h1>
    <h2 style="color:red">这是一封 HTML 测试邮件</h2>
    """
    sendMail(body)
```

监控 ☆

发件人: baojingtongzhi <baojingtongzhi@163.com> 

时 间: 2020年9月14日 (星期一) 下午2:09

收件人: 可爱ノ的你' <1121267855@qq.com>

测试邮件

这是一封HTML测试邮件

示例 5: 发送图片邮件

```
import smtplib
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.header import Header
from email.utils import parseaddr, formataddr

# 格式化邮件地址
def formatAddr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))

def sendMail(body, image):
    smtp_server = 'smtp.163.com'
    email_account = 'baojingtongzhi@163.com'
    account_password = 'VCGXTBHXPVDVLGZ'
    to_mail = ["1121267855@qq.com"]

    # 构造一个 MIMEMultipart 对象代表邮件本身
    msg = MIMEMultipart()
    # Header 对中文进行转码
    msg['From'] = formatAddr('管理员 <%s>' % email_account)
    msg['To'] = ', '.join(to_mail)
    msg['Subject'] = Header('监控', 'utf-8')
    msg.attach(MIMEText(body, 'html', 'utf-8'))

    # 二进制模式读取图片
    with open(image, 'rb') as f:
        msgImage = MIMEImage(f.read()) # 使用 MIMEImage 类构造图片并添加到 msg 对象
        # 定义图片 ID, 根据 ID 在 HTML 里获取图片
        msgImage.add_header('Content-ID', '<image1>')
        msg.attach(msgImage)
```

```
try:
    s = smtplib.SMTP()
    s.connect(smtplib.SMTP_SERVER, "25")
    s.login(email_account, account_password)
    s.sendmail(email_account, to_mail, msg.as_string()) # as_string()把MIMEText对象变成str
    s.quit()
except smtplib.SMTPException as e:
    print("Error: %s" % e)
if __name__ == "__main__":
    body = """
    <h1>测试图片</h1>
    
    """
    sendMail(body, "123.jpg")
```

监控 ☆

发件人: **baojingtongzhi** <baojingtongzhi@163.com> 

时 间: 2020年9月14日 (星期一) 下午2:26

收件人: 可爱ノ的你' <1121267855@qq.com>

测试图片



第十二章 Python 正则表达式

正则表达式是对字符串操作的一种逻辑方式，就是用实现定义好的一些特定字符及这些特定字符的组合，组成一个规则字符串，这个规则字符串就是表达对字符串的逻辑，给定一个正则表达式和另一个字符串，通过正则表达式从字符串我们想要的部分。

正则表达式简单来说就是通过一个规则来匹配想要的字符串。正则表达式在每种语言中都会有，用法也基本一样。

Python 正则表达式主要由 re 库提供，拥有了基本所有的表达式。

re 模块有以下常用的方法：

方法	描述
re.compile(pattern, flags=0)	把正则表达式编译成一个对象
re.match(pattern, string, flags=0)	匹配字符串开始，如果不匹配返回 None
re.search(pattern, string, flags=0)	扫描字符串寻找匹配，如果符合返回一个匹配对象并终止匹配，否则返回 None
re.split(pattern, string, maxsplit=0, flags=0)	以匹配模式作为分隔符，切分字符串为列表
re.findall(pattern, string, flags=0)	以列表形式返回所有匹配的字符串
re.finditer(pattern, string, flags=0)	以迭代器形式返回所有匹配的字符串
re.sub(pattern, repl, string, count=0, flags=0)	字符串替换，repl 替换匹配的字符串，repl 可以是一个函数

12.1 re 模块的基本使用

re.compile()

语法：re.compile(pattern, flags=0)
pattern 指的是正则表达式。flags 是标志位的修饰符，用于控制表达式匹配模式

```
import re
s = "this is test string"
pattern = re.compile('this')
result = pattern.match(s)
print(result)
```

或者直接使用 re.mach

```
result = re.match('this', s)
print(result)
```

前者好处是先使用 compile 把正则表达式编译一个对象，方便再次使用。后者更直观。

match()

语法：re.match(pattern, string, flags=0)

例如：判断字符串开头是否匹配字符

```
import re
s = "hello world!"
result = re.match("hello", s)
print(result)
# 运行结果
<re.Match object; span=(0, 5), match='hello'>
```

匹配成功后，result 对象会增加一个 group() 方法，可以用它来获取匹配结果：

```
print(result.group())
# 运行结果
hello
```

匹配失败，就不能使用 group() 方法了，会提示没属性：

```
result = re.match("world", s)
print(result)
print(result.group())
# 运行结果
None
AttributeError: 'NoneType' object has no attribute 'group'
```

这时就可以对 result 对象进行判断：

```
if result:
    print("匹配结果：%s" %result.group())
else:
    print("匹配失败")
```

小结：match() 是从左向右匹配，所以在写正则表达式时应考虑到。

12.2 代表字符

字符	描述
.	任意单个字符（除了\n）
[]	匹配中括号中的任意单个字符。并且特殊字符写在[]会被当成普通字符来匹配
[.-.]	匹配中括号中范围内的任意单个字符，例如[a-z], [0-9]
[^]	匹配[^字符]之外的任意单个字符
\d	匹配数字，等效[0-9]
\D	匹配非数字字符，等效[^0-9]

<code>\s</code>	匹配单个空白字符（空格、Tab 键），等效 <code>[\t\n\r\f\v]</code>
<code>\S</code>	匹配空白字符之外的所有字符，等效 <code>[^\t\n\r\f\v]</code>
<code>\w</code>	匹配字母、数字、下划线，等效 <code>[a-zA-Z0-9_]</code>
<code>\W</code>	与 <code>\w</code> 相反，等效 <code>[^a-zA-Z0-9_]</code>

示例：匹配单个字符

```
import re
s = "hello world!"
result1 = re.match(".", s)
result2 = re.match("..", s)
result3 = re.match("...", s)
print(result1.group())
print(result2.group())
print(result3.group())
# 运行结果
h
he
hel
```

示例：匹配任意单字符

```
import re
s = "hello world"
result = re.match("[h]", s)
print(result)
result = re.match("[abch]", s)
print(result)
result = re.match("[ahc]", s)
print(result)
result = re.match("[^abc]", s)
print(result)
result = re.match("[a-z]", s)
print(result)
result = re.match("[a-z][a-z][a-z][a-z][a-z]", s)
print(result)

# 执行结果
<re.Match object; span=(0, 1), match='h'>
<re.Match object; span=(0, 1), match='h'>
<re.Match object; span=(0, 1), match='h'>
<re.Match object; span=(0, 1), match='h'>
<re.Match object; span=(0, 5), match='hello'>
```

只要是中括号字符包含字符串的第一个字符就可以匹配到。

示例：匹配数字


```
s = "13812345678"
result = re.match("\d\d\d\d\d", s)
print(result)
# 运行结果
<re.Match object; span=(0, 5), match='13812'>
```

一个\d 就匹配一个数字。

示例：匹配空格

```
s = "138 1234 5678"
result = re.match("\d\d\d\d\d", s)
print(result)
# 运行结果
None
```

匹配不到，这个因为第四个字符是空格，而\d 只匹配数字。
可以使用\s 匹配空格：

```
result = re.match("\d\d\d\s\d", s)
print(result)
# 运行结果
<re.Match object; span=(0, 5), match='138 1'>
```

示例：匹配字母、数字、下划线

```
s = "aB1_"
result = re.match("\w", s)
print(result)
result = re.match("\w\w", s)
print(result)
result = re.match("\w\w\w", s)
print(result)
result = re.match("\w\w\w\w", s)
print(result)
result = re.match("\w\w\w\w\w", s)
print(result)

# 执行结果
<re.Match object; span=(0, 1), match='a'>
<re.Match object; span=(0, 2), match='aB'>
<re.Match object; span=(0, 3), match='aB1'>
<re.Match object; span=(0, 4), match='aB1_'>
None
```

12.3 原始字符串符号“r”

r 表示原始字符串，有了它，字符串里的特殊意义符合就会自动加转义符。

```
import re
s = "123\\abc"
result = re.match("123\\abc", s)
print(result)
# 运行结果
None
```

同样的字符串，为啥匹配不到呢？因为正则表达式里\\其中一个作为了转义符，实际与字符串不一样。既然这样，我们可以再为\转义：

```
result = re.match("123\\\\abc", s)
print(result)
# 运行结果
<re.Match object; span=(0, 7), match='123\\abc'>
```

可以匹配到，但这样做有点麻烦，这时可以用 r 字符很方便解决：

```
result = re.match(r"123\\abc", s)
print(result)
```

实际上，r 字符也是自动帮你加了\转义符，这个效果可以在解释器里看到：

```
>>> s = r"123\\abc"
>>> s
'123\\\\abc'
```

12.4 代表数量

字符	描述
*	匹配前面的子表达式零次或多次
+	匹配前面的子表达式一次或多次
?	匹配前面的子表达式零次或一次
{n} 或 {n, }	匹配花括号前面字符至少 n 个字符
{n, m}	匹配花括号前面字符至少 n 个字符，最多 m 个字符

示例：匹配固定次数

```
import re
s = "hello world"
result = re.match("hel.*", s) # .*匹配所有
print(result)
result = re.match("hel*", s) # 匹配前面 l 字符一次
print(result)
```

```

result = re.match("hel6*", s) # 允许前面字符没有匹配到
print(result)
result = re.match("hel+", s) # 匹配前面 1 字符一次
print(result)
result = re.match("hel6+", s) # 至少前面 6 数字一次，但没匹配到
print(result)
result = re.match("hel?", s) # 只能匹配前面 1 字符一次，如果两次无效
print(result)
result = re.match("hel6?", s) # 允许前面字符没有匹配到
print(result)
# 运行结果
<re.Match object; span=(0, 11), match='hello world'>
<re.Match object; span=(0, 4), match='hell'>
<re.Match object; span=(0, 3), match='hel'>
<re.Match object; span=(0, 4), match='hell'>
None
<re.Match object; span=(0, 3), match='hel'>
<re.Match object; span=(0, 3), match='hel'>

```

示例：匹配指定次数

```

import re
s = "hello world"
result = re.match(".{5}", s) # 匹配前 5 个字符
print(result)
result = re.match(".{3,8}", s) # 匹配至少 3 个字符，最多 8 个字符
print(result)
s = "13812345678"
result = re.match("\d{11}", s) # 匹配 11 位数字
print(result)
s = "1381234abcd"
result = re.match("\d{11}", s) # 不满足 11 位数字
print(result)
s = "138123456789abc"
result = re.match("\d{11}", s) # 匹配 11 位数字，其他字符不匹配
print(result)
# 运行结果
<re.Match object; span=(0, 5), match='hello'>
<re.Match object; span=(0, 8), match='hello wo'>
<re.Match object; span=(0, 11), match='13812345678'>
None
<re.Match object; span=(0, 11), match='13812345678'>

```

综合示例：判断用户输入邮箱格式是否正确

```

import re
email = input("请输入你的邮箱：")
result = re.match("\w{4,20}@[0-9]+\.[a-z]+$", email)
if result:
    print("邮箱格式正确，你输入的邮箱是：%s" % result.group())

```

```
else:
    print("邮箱格式错误")
# 运行结果
请输入你的邮箱: aliang@163.com
邮箱格式正确, 你输入的邮箱是: aliang@163.com
```

将不固定的通过特定符合指定。

12.5 代表边界

字符	描述
<code>^</code>	匹配字符串开头
<code>\$</code>	匹配字符串结尾
<code>\b</code>	匹配单词边界
<code>\B</code>	匹配非单词边界

再完善下上面匹配邮箱示例，可能会出现这种情况：

```
请输入你的邮箱: _aliang@163.com    # 开头多输入一个感叹号
邮箱格式正确, 你输入的邮箱是: _aliang@163.com
请输入你的邮箱: aliang@163.com.    # 结尾多输入一个点
邮箱格式正确, 你输入的邮箱是: aliang@163.com
```

开头结尾可能输错了，还是验证成功，实际是不对的，这里需要进一步限制下：

```
import re
email = input("请输入你的邮箱: ")
result = re.match("^[a-z]\w{4,20}@[0-9]+\.[a-z]+$", email)
if result:
    print("邮箱格式正确, 你输入的邮箱是: %s" % result.group())
else:
    print("邮箱格式错误")
# 执行结果
请输入你的邮箱: _aliang@163.com
邮箱格式错误
请输入你的邮箱: aliang@163.com.
邮箱格式错误
```

使用`^`和`$`分配限制了开头和结尾必须是`a-z`字母。

12.6 代表分组

字符	描述
	匹配竖杠两边的任意一个正则表达式
(re)	匹配小括号中正则表达式。 使用\n 反向引用，n 是数字，从 1 开始编号，表示引用第 n 个分组匹配的内容。
(?P<name>re)	命名分组，name 是标识名称，默认是索引标识分组匹配的内容

继续看上面匹配邮箱示例，其实还有个问题要解决，例如邮箱运营商也只是 163，还有 qq、126 等，所以也要能匹配其他家的邮箱地址。

运行结果：

```
请输入你的邮箱：aliang@qq.com
邮箱格式错误
```

修改正则表达式：

```
import re
email = input("请输入你的邮箱：")
result = re.match("^[a-z]\w{4,20}@(163|126|qq)+\.[a-z]+$", email)
if result:
    print("邮箱格式正确, 你输入的邮箱是： %s" %result.group())
else:
    print("邮箱格式错误")
# 运行结果
请输入你的邮箱：aliang@qq.com
邮箱格式正确, 你输入的邮箱是： aliang@qq.com
```

(|)作用是匹配其中任意一个表达式。并且也不会捕获分组里面的值。

```
s = "<h1>hello world</h1>"
result = re.match("<(\w+)>.*</(\w+)>", s) #这里面两个分组，一个()表示一个分组，并且会保存匹配的内容
print(result)
result = re.match(r"<(\w+)>.*</(\1)>", s) # 使用\n 反向引用第一个分组，相当于使用第一个()中的表达式。
print(result)
# 运行结果
<re.Match object; span=(0, 20), match='<h1>hello world</h1>'>
<re.Match object; span=(0, 20), match='<h1>hello world</h1>'>
```

进一步验证\n 反向引用作用：

```
s = "<h1>hello world</h2>" # 有意将/h1 修改为 h2
result = re.match(r"<(\w+)>.*</(\1)>", s) # 匹配失败，使用第一个()中的表达式是匹配不上
```

```
h2 的
print(result)
# 运行结果
None
```

在使用\n 反向引用时需要使用过使用原始字符串 r 来为\1 中的\转义或者直接写\\1
当使用分组匹配时，group(index) 方法可以指定获取指定分组匹配的内容：

```
import re
s = "hello world!"
result = re.match("(\w+) (\w+)", s)
print("所有分组匹配结果：%s" %result.group())
print("第一个括号匹配结果：%s" %result.group(1))
print("第二个括号匹配结果：%s" %result.group(2))
print("元组形式返回所有组：", result.group(1, 2))
# 运行结果
所有分组匹配结果：hello world
第一个括号匹配结果：hello
第二个括号匹配结果：world
元组形式返回所有组： ('hello', 'world')
```

分组作用：

- 引用分组
- 可以取指定分组匹配的内容

当分组使用较多时，\n 反向引用容易出错，这时可以给分组命名，通过名称引用分组：

正则里引用分组：

```
s = "<h1>hello world</h1>"
result = re.match("<(P<g1>\w+)>.*</(P=g1)>", s) # 正则里引用分组
print(result)
```

groups 方法通过名称获取分组匹配内容：

```
s = "hello world"
result = re.match("(?P<h>\w+) (?P<w>\w+)", s)
print("w 分组匹配结果：%s" %result.group("w"))
print("h 分组匹配结果：%s" %result.group("h"))
```

还有一些不常用的分组字符：

(?#re)	注释小括号内的内容，提供注释功能
(?:re)	不保存匹配的分组，也不分配组号
(?=re)	匹配 re 前面的内容，称为正先行断言
(?!re)	匹配后面不是 re 的内容，称为负先行断言

(?<=re)	匹配 re 后面的内容，称为正后发断言
(?<!re)	匹配前面不是 re 的内容，称为负后发断言
(?(id/name)Y/N)	如果分组提供的 id 或 name 存在，则使用 Y 表达式匹配，否则 N 表达式匹配

12.7 贪婪和非贪婪匹配

贪婪模式：尽可能最多匹配

非贪婪模式，尽可能最少匹配，一般在量词（*、+）后面加个？问号就是非贪婪模式。

示例 1：

```
s = "hello 666666"
result = re.match("hello 6+", s) # 贪婪匹配
print(result)
result = re.match("hello 6+?", s) # 非贪婪匹配
print(result)
# 运行结果
<re.Match object; span=(0, 12), match='hello 666666'>
<re.Match object; span=(0, 7), match='hello 6'>
```

示例 2：

```
s = "<div>1</div><div>2</div>"
result = re.match("<div>.*</div>", s)
print(result)
# 运行结果
<re.Match object; span=(0, 24), match='<div>1</div><div>2</div>'>
```

匹配所有，但是我只想取第一个 div：

```
s = "<div>1</div><div>2</div>"
result = re.match("<div>.*?</div>", s) # 正则匹配到就截止
print(result)
# 运行结果
<re.Match object; span=(0, 12), match='<div>1</div>'>
```

示例 3：我想取其中年龄数字

```
s = "我今年 30 岁，身高 185cm"
result = re.match("\w+(\d+)\w+", s)
print(result.group(1))
# 运行结果
0
```

发现结果不是我们想要的，这是因为第一个 \w+ 会尽可能匹配，截止到下一个 \d+，即数字 3 也在其范围。想解决这个问题也简单，只需要把要获取的内容前面正则设置非贪婪匹配：

```
result = re.match("\w+(\d+)\w+", s)
print(result.group(1))
# 运行结果
30
```

小结：贪婪匹配是尽可能的向右匹配，直到字符串结束。非贪婪匹配是匹配满足后就结束。

12.8 其他方法

search()

match()是从左向右匹配，search()只要匹配到就返回。
上面取年龄示例如果用这个方法就方便很多了：

```
s = "我今年 30 岁，身高 185cm"
result = re.search("\d+", s)
print(result.group())
# 运行结果
30
```

findall()

findall()方法用于将匹配所有的结果以列表形式

```
s = "hello world"
result = re.findall("o", s)
print(result)
```

对应的还有个 finditer() 方法，返回一个迭代器。

split()

split()方法用于分隔字符，以列表形式返回

```
s = "我今年 30 岁，身高 185cm"
result = re.split("\d+", s)
print(result)
# 运行结果
['我今年', '岁，身高', 'cm']
```

sub()

sub()方法用于替换字符。

```
result = re.sub("30", "31", s)
print(result)
```



```
# 运行结果
我今年 31 岁，身高 185cm
```

12.9 标志位

字符	描述
re.I/re.IGNORECASE	忽略大小写
re.S/re.DOTALL	匹配所有字符，包括换行符\n，如果没这个标志将匹配除了换行符
re.DEBUG	显示关于编译正则的 debug 信息
re.L/re.LOCALE	本地化匹配，影响\w, \W, \b, \B, \s 和\S
re.M/re.MULTILINE	多行匹配，影响^和\$
re.U/re.UNICODE	根据 unicode 字符集解析字符。影响影响\w, \W, \b, \B, \d, \D, \s 和\S
re.X/re.VERBOSE	允许编写更好看、更可读的正则表达式，也可以在表达式添加注释

可能会用的前面两个，举例说明。

忽略大小写：

```
s = "hello world"
result = re.match("Hello", s)
print(result)
result = re.match("Hello", s, re.I) # 忽略大小写可以匹配到
print(result)
# 运行结果
None
<re.Match object; span=(0, 5), match='hello'>
```

匹配换行符：

```
s = """hello
world
"""

result = re.match("hello.*", s) # 只匹配到 hello，即使使用.*也不行
print(result)
result = re.match("hello.*", s, re.S) # 可以匹配所有
print(result)
# 运行结果
<re.Match object; span=(0, 5), match='hello'>
<re.Match object; span=(0, 12), match='hello\nworld\n'>
```

第十三章 Python 数据库编程

本章节学习 Python 操作数据库，完成简单的增删改查工作，以 MySQL 数据库为例。

13.1 PyMySQL

pymysql 是 Python 中操作 MySQL 的模块，其使用方法和 MySQLdb 几乎相同。但目前 pymysql 支持 python3.x 而后者不支持 3.x 版本。

13.1.1 安装 pymysql 模块

pymysql 是第三方模块，需要单独安装，首选通过 pip 安装 PyMySQL：

```
# pip3 install pymysql
```

13.1.2 pymysql 使用

官方文档：<https://pypi.org/project/PyMySQL/#documentation>

根据官方文档做一个示例，先登录 Mysql 实例创建一个 test 库和一张 users 表，如下：

```
[root@localhost ~]# mysql -uroot -p
mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> CREATE TABLE `users` (
  ->   `id` int(11) NOT NULL AUTO_INCREMENT,
  ->   `email` varchar(255) COLLATE utf8_bin NOT NULL,
  ->   `password` varchar(255) COLLATE utf8_bin NOT NULL,
  ->   PRIMARY KEY (`id`)
  -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin
  -> AUTO_INCREMENT=1 ;
Query OK, 0 rows affected (0.11 sec)
```

将官方文档复制到 db.py 文件里并运行：

```
import pymysql
connection = pymysql.connect(host='localhost',
                             user='root',
                             password='123456',
                             db='test',
                             charset='utf8mb4',
                             cursorclass=pymysql.cursors.DictCursor)

try:
    with connection.cursor() as cursor:
        # 创建一条记录
```

```

sql = "INSERT INTO `users` (`email`, `password`) VALUES (%s, %s)"
cursor.execute(sql, ('webmaster@python.org', 'very-secret'))

# 默认情况下，连接不会自动提交，必须手动提交保存
connection.commit()

with connection.cursor() as cursor:
    # 读取一条记录
    sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"
    cursor.execute(sql, ('webmaster@python.org',))
    result = cursor.fetchone()
    print(result)
finally:
    connection.close()
# 运行结果
{'id': 1, 'password': 'very-secret'}
```

执行成功，进 Mysql 看下是否插入成功：

```

[root@localhost ~]# mysql -uroot -p
mysql> use test;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from users;
+----+-----+-----+
| id | email                | password |
+----+-----+-----+
| 1  | webmaster@python.org | very-secret |
+----+-----+-----+
1 row in set (0.00 sec)
```

可以看到成功插入了记录。

代码说明：先创建一个连接对象，下面是一个异常处理，如果连接成功，执行 INSERT 语句并提交，再执行 SELECT 语句，获取查询的第一条记录并打印。finally 是最终执行关闭连接。

在上面示例中用到了 pymysql.connect() 函数，还有游标对象 cursor，下面具体看下它们的用法。

connect() 函数常用参数：

方法	描述
host	数据库主机地址
user	数据库账户
passwd	账户密码
db	使用的数据库
port	数据库主机端口，默认 3306
connect_timeout	连接超时时间，默认 10，单位秒

方法	描述
charset	使用的字符集
cursorclass	自定义游标使用的类。上面示例用的是字典类，以字典形式返回结果，默认是元组形式。

连接对象常用方法：

方法	描述
commit()	提交事务。对支持事务的数据库和表，如果提交修改操作，不适用这个方法，则不会写到数据库中
rollback()	事务回滚。对支持事务的数据库和表，如果执行此方法，则回滚当前事务。在没有 commit() 前提下。
cursor([cursorclass])	创建一个游标对象。所有的 sql 语句的执行都要在游标对象下进行。MySQL 本身不支持游标，MySQLdb 模块对其游标进行了仿真。

游标对象常用方法：

方法	描述
close()	关闭游标
execute(sql)	执行 sql 语句
executemany(sql)	执行多条 sql 语句
fetchone()	从运行结果中取第一条记录
fetchmany(n)	从运行结果中取 n 条记录
fetchall()	从运行结果中取所有记录

13.1.3 数据库增删改查

了解了 pymysql 基本用法，下面进一步对上面方法演示，以数据库增删改查为讲解思路。

(1) 增

在 user 表里再添加一条记录：

```
>>> import pymysql
>>> connection = pymysql.connect(host='localhost',
...                               user='root',
...                               password='123456',
...                               db='test',
...                               charset='utf8mb4',
...                               cursorclass=pymysql.cursors.DictCursor)
>>> cursor = connection.cursor()
>>> sql = "INSERT INTO `users` (`email`, `password`) VALUES (%s, %s)"
```

```
>>> cursor.execute(sql, ('user1@python.org', '123456'))
1      # 影响的行数
>>> connection.commit()      # 提交事务，写到数据库
>>> sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"
>>> cursor.execute(sql, ('user1@python.org',))
1
>>> result = cursor.fetchone()
>>> print(result)
{'id': 2, 'password': '123456'}
```

插入多条记录:

```
>>> sql = "INSERT INTO `users` (`email`, `password`) VALUES (%s, %s)"
>>> args = [('user2@python.org', '123456'), ('user3@python.org', '123456'), ('user4@python.org', '123456')]
>>> cursor.executemany(sql, args)
3
>>> connection.commit()
>>> sql = "SELECT `id`, `email`, `password` FROM `users`"
>>> cursor.execute(sql)
5
>>> result = cursor.fetchall()
>>> print(result)
[{'id': 1, 'email': 'webmaster@python.org', 'password': 'very-secret'}, {'id': 2, 'email': 'user1@python.org', 'password': '123456'}, {'id': 3, 'email': 'user2@python.org', 'password': '123456'}, {'id': 4, 'email': 'user3@python.org', 'password': '123456'}, {'id': 5, 'email': 'user4@python.org', 'password': '123456'}]
```

args 变量是一个包含多元组的列表，每个元组对应着每条记录。当查询多条记录时，使用此方法，可有效提高插入效率。

(2) 查

查询 users 表记录:

```
>>> sql = "SELECT `id`, `email`, `password` FROM `users`"
>>> cursor.execute(sql)
5
>>> cursor.fetchone()      # 获取第一条记录
{'id': 1, 'email': 'webmaster@python.org', 'password': 'very-secret'}
>>> cursor.execute(sql)
5
>>> cursor.fetchmany(2)     # 获取前两条记录
[{'id': 1, 'email': 'webmaster@python.org', 'password': 'very-secret'}, {'id': 2, 'email': 'user1@python.org', 'password': '123456'}]
>>> cursor.fetchall()      # 获取所有记录
[{'id': 3, 'email': 'user2@python.org', 'password': '123456'}, {'id': 4, 'email': 'user3@python.org', 'password': '123456'}, {'id': 5, 'email': 'user4@python.org', 'password': '123456'}]
```

(3) 改

将 user1@python.org 密码修改为 456789:

```
>>> sql = "UPDATE users SET `password`='456789' WHERE `email`='user1@python.org'"
>>> cursor.execute(sql)
1
>>> sql = "SELECT `id`, `email`, `password` FROM `users`"
>>> cursor.execute(sql)
5
>>> cursor.fetchmany(2)
[{'id': 1, 'email': 'webmaster@python.org', 'password': 'very-secret'}, {'id': 2, 'email': 'user1@python.org', 'password': '456789'}]
```

(4) 删

删除 email 是 webmaster@python.org 的记录:

```
>>> sql = 'DELETE FROM `users` WHERE email="webmaster@python.org"'
>>> cursor.execute(sql)
1
>>> connection.commit()
>>> sql = "SELECT `id`, `password` FROM `users` WHERE `email`=%s"
>>> cursor.execute(sql, ('webmaster@python.org',))
0
>>> sql = "SELECT `id`, `email`, `password` FROM `users`"
>>> cursor.execute(sql)
4
>>> cursor.fetchall()
[{'id': 2, 'email': 'user1@python.org', 'password': '123456'}, {'id': 3, 'email': 'user2', 'password': '123456'}, {'id': 4, 'email': 'user3', 'password': '123456'}, {'id': 5, 'email': 'user4', 'password': '123456'}]
```

13.1.4 遍历查询结果

```
[root@localhost ~]# cat pymysql-1.py
#!/usr/local/bin/python3.6
import pymysql
connection = pymysql.connect(host='localhost',
                             user='root',
                             password='123456',
                             db='test',
                             charset='utf8mb4',
                             cursorclass=pymysql.cursors.DictCursor)

try:
    with connection.cursor() as cursor:
        sql = "SELECT `id`, `email`, `password` FROM `users`"
        cursor.execute(sql)
        result = cursor.fetchall()
        for dict in result:
            print("email:%s, password:%s" %(dict['email'],dict['password']))
finally:
    connection.close()
```

```
[root@localhost ~]# python3.6 pymysql-1.py
email:webmaster@python.org, password:very-secret
email:user1@python.org, password:123456
email:user2@python.org, password:123456
email:user3@python.org, password:123456
email:user4@python.org, password:123456
```

小结：pymysql 模块主要为我们提供连接数据库和获取执行 sql 结果的方法。

13.2 SQLAlchemy

官方文档：<http://docs.sqlalchemy.org/en/latest/contents.html>

在 Python 中，最有名的 ORM 框架是 SQLAlchemy。

什么是 ORM？

ORM (Object-Relational Mapping, 对象关系映射)：是一种程序技术，用于将关系数据库的表结构映射到对象上。ORM 提供了概念性、易于理解的模型化的方法。

为什么出现 ORM 技术？

一句话：为了避免写复杂的 sql 语句。

13.2.1 安装 sqlalchemy 模块

sqlalchemy 也是第三方模块，需要单独安装，通过 pip 安装 SQLAlchemy：

```
# pip3 install sqlalchemy
```

13.2.2 基本使用

使用 SQLAlchemy 模块大致分为以下四步：

第一步 导入 SQLAlchemy 模块指定类，并创建 Mysql 连接

```
>>> from sqlalchemy import create_engine
```

连接 MySQL 数据库，使用 create_engine()：

```
>>> engine = create_engine('mysql+pymysql://root:123456@localhost:3306/test')
```

格式：'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'

此时 engine 变量是一个实例，它表示数据库的核心接口。create_engine() 创建一个数据库连接时，它尚未打开连接。当第一次调用时，才会由 Engine 维护的连接池建立一个连接，并保留它，直到我们提交所有更改或关闭会话对象。

第二步 声明映射

当使用 ORM 时，首选描述我们要处理的数据库表，通过定义自己的类到这些表。

映射表是通过一个基类来定义的，所以要先导入它：

```
>>> from sqlalchemy.ext.declarative import declarative_base
>>> Base = declarative_base()
```

现在有了一个基类，我们可以根据它定义任意数量的映射类。例如定义数据库表名为 users，该类中定义了表的详细信息，主要是表名，以及列的名称和数据类型：

```
>>> from sqlalchemy import Column, Integer, String
>>> class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    email = Column(String(255))
    password = Column(String(255))
```

一个映射类至少需要一个__tablename__属性，并且至少有一个 Column 是主键。

如果多个表，就继续定义其他表的类，跟上面代码一样。

第三步 创建会话

现在开始与数据库交互，先创建一个会话：

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)
>>> Session = sessionmaker()
>>> Session.configure(bind=engine)
```

实例化一个 Session：

```
>>> session = Session()
```

第四步 增删改查

添加对象，有了 ORM，我们向数据库表中添加一行记录，可以视为添加一个 User 对象：

```
>>> add_user = User(email='user5@python.org', password='123456')
>>> session.add(add_user)
>>> session.commit()
```

创建 Query 查询，filter 是 where 条件，最后调用返回唯一行，如果调用 all() 则返回所有行，并且以列表类型返回：

```
>>> user = session.query(User).filter(User.email=='user5').one()
>>> user.email
'user5@python.org'
>>> user.password
'123456'
```

可以给 User 一次添加多个对象，使用 add_all()：

```
>>> session.add_all([
...     User(email='user6@python.org', password='123456'),
...     User(email='user7@python.org', password='123456'),
...     User(email='user8@python.org', password='123456')])
```

有三个新 User 对象正在等待处理：

```
>>> session.new
IdentitySet([<__main__.User object at 0x7fa80bd71ba8>, <__main__.User object at 0x7fa80bd71908>, <__main__.User object at 0x7fa80bd716d8>])
```

将更改刷新到数据库，并提交事务。


```
>>> session.commit()
```

修改数据：

```
>>> session.query(User).filter(User.email=='user5@python.org').update({"password":456789})
1
>>> user = session.query(User).filter(User.email=='user5@python.org').one()
>>> user.password
'456789'
>>> session.commit()
```

再总结下整体步骤：

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import sessionmaker

# 创建对象的基类
Base = declarative_base()

# 定义 User 对象
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    email = Column(String(255))
    password = Column(String(255))

# 初始化数据库连接
engine = create_engine('mysql+pymysql://root:123456@localhost:3306/test')
# 创建 Session 类型
Session = sessionmaker(bind=engine)
Session = sessionmaker()
Session.configure(bind=engine)
session = Session()

# 查询所有记录
user = session.query(User).all()
for i in user:
    print("email: %s, password: %s" % (i.email, i.password))
```

运行结果：

```
email: user1@python.org, password: 123456
email: user2@python.org, password: 123456
email: user3@python.org, password: 123456
email: user4@python.org, password: 123456
email: user5@python.org, password: 123456
email: user6@python.org, password: 456789
```

```
email: user7@python.org, password: 123456
email: user8@python.org, password: 123456
```

当我们查询一个 User 对象时，该对象的返回一个包含若干个的对象列表。

小结：ORM 框架的作用就是把数据库表的一行记录与一个对象相互做自动转换。

第十四章 Python 批量管理主机

随着业务量增加，服务器也会越来越多，如果高效管理上百台服务器是一个重要的问题。运维在管理服务器时经常有这么一个需求：统计这些服务器基本信息，例如 CPU 几核的，内存多少 G、磁盘多大的等等，如果手动完成工作量会很大，这时就得考虑编写脚本自动化实现，在 Python 提供了 paramiko 模块可以很方便实现批量 Linux 服务器执行 Shell 命令。

Python 也提供了 pexpect 模块可自动化交互执行命令，例如网络设备、FTP 服务器等。

14.1 paramiko

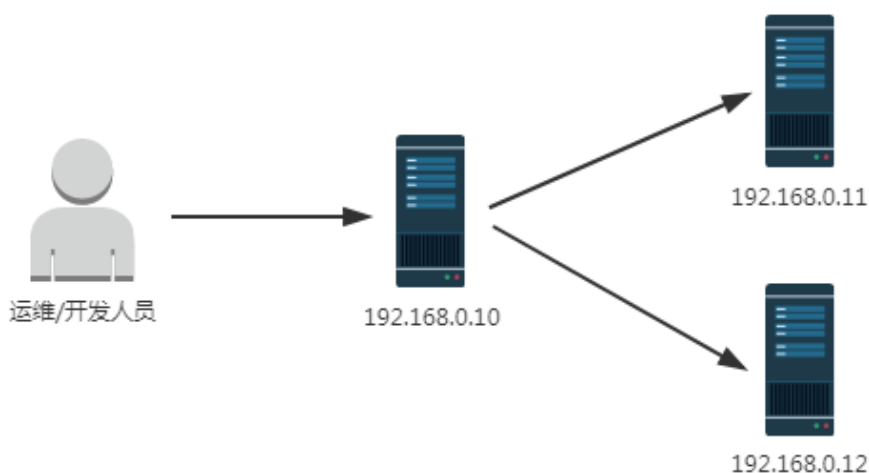
paramiko 模块是基于 Python 实现的 SSH 远程安全连接，用于 SSH 远程执行命令、文件传输等功能。首先 pip 安装：

```
pip3 install paramiko
```

如果安装失败，也使用 yum 安装：

```
yum install python-paramiko
```

为更好学习该模块，我们下面写几个具体的示例来熟悉它的常用用法。
拓扑图：



14.1.1 SSH 密码认证远程执行命令

```
import paramiko
hostname = '192.168.0.11'
port = 22
username = 'root'
password = '123456'

# 绑定实例
client = paramiko.SSHClient()
# AutoAddPolicy() 自动添加主机 keys
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
# 连接主机信息
client.connect(hostname, port, username, password, timeout=5)
# 执行 Shell 命令，结果分别保存在标准输入，标准输出和标准错误
stdin, stdout, stderr = client.exec_command('ls -l')
stdout = stdout.read()
error = stderr.read()
# 判断 stderr 输出是否为空，为空则打印运行结果，不为空打印报错信息
if not error:
    print(result)
else:
    print(error)

client.close()

# 运行结果
b'total 0\n-rw-rw-r-- 1 root root 0 Jun 28 19:26 123.txt\n'
```

14.1.2 SSH 密钥认证远程执行命令

口令是普遍的鉴权策略，为了提高安全性，还会用密钥对认证。
首选生成密钥对：

```
# ssh-keygen # 采用默认配置，一直回车即可
# ls .ssh/
id_rsa id_rsa.pub
```

将 id_rsa.pub 公钥追加到目标服务器/root/.ssh/authorized_keys 文件中。

```
import paramiko
import sys
hostname = '192.168.0.11'
port = 22
username = 'root'
key_file = '/root/.ssh/id_rsa'
# 将列表元素以空格拼接
```

```

cmd = " ".join(sys.argv[1:])
def ssh_conn(command):
    client = paramiko.SSHClient()
    # 指定 key 文件
    key = paramiko.RSAKey.from_private_key_file(key_file)
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    # 使用 key 登录
    client.connect(hostname, port, username, pkey=key)
    stdin, stdout, stderr = client.exec_command(command)
    result = stdout.read()
    error = stderr.read()
    if not error:
        print(result)
    else:
        print(error)
    client.close()
if __name__ == "__main__":
    ssh_conn(cmd)

# 运行结果
b'total 0\n-rw-rw-r-- 1 root root 0 Jun 28 19:26 123.txt\n'

```

14.1.3 上传文件到远程服务器

```

import os, sys
import paramiko

hostname = '192.168.0.11'
port = 22
username = 'root'
password = '123456'
local_path = '/root/test.txt'
remote_path = '/opt/test.txt'
if not os.path.isfile(local_path):
    print(local_path + " 文件不存在!")
    sys.exit(1)
try:
    s = paramiko.Transport((hostname, port))
    s.connect(username=username, password=password)
except Exception as e:
    print(e)
    sys.exit(1)
sftp = paramiko.SFTPClient.from_transport(s)
# 使用 put() 方法把本地文件上传到远程服务器
sftp.put(local_path, remote_path)
# 简单测试是否上传成功
try:

```

```

    # 如果远程主机有这个文件则返回一个对象，否则抛出异常
    sftp.file(remote_path)
    print("上传成功.")
except IOError:
    print("上传失败!")
finally:
    s.close()

# 运行结果
上传成功.

```

14.1.4 上传目录到远程服务器

paramiko 模块并没有实现直接上传目录的类，上述已经实现了上传文件，再写一个上传目录的代码就简单了，利用 os 库的 os.walk() 方法遍历目录，再一个个上传：

```

import os, sys
import paramiko

hostname = '192.168.0.11'
port = 22
username = 'root'
password = '123456'
local_path = '/root/test'
remote_path = '/opt/test'

# 如果路径末尾带"/"就去除
if local_path[-1] == '/':
    local_path = local_path[0:-1]
if remote_path[-1] == '/':
    remote_path = remote_path[0:-1]

# 递归将本地目录所有文件以绝对路径存储在列表
file_list = []
if os.path.isdir(local_path):
    for root, dirs, files in os.walk(local_path):
        for file in files:
            # 获取文件绝对路径
            file_path = os.path.join(root, file)
            file_list.append(file_path)
else:
    print(path + "目录不存在!")
    sys.exit(1)

try:
    s = paramiko.Transport((hostname, port))
    s.connect(username=username, password=password)
    sftp = paramiko.SFTPClient.from_transport(s)

```

```

except Exception as e:
    print(e)

# 遍历列表
for local_file in file_list:
    # 替换目标目录
    remote_file = local_file.replace(local_path, remote_path)
    remote_dir = os.path.dirname(remote_file)
    # 如果远程服务器没目标目录则创建
    try:
        sftp.stat(remote_dir)
    except IOError:
        sftp.mkdir(remote_dir)
    print("%s -> %s" % (local_file, remote_file))
    sftp.put(local_file, remote_file)
s.close()

```

现在本地创建目录和测试文件：

```

[root@localhost ~]# mkdir test
[root@localhost ~]# touch test{1..3}.txt
[root@localhost ~]# python test.py
/root/test/test1.txt -> /opt/test/test1.txt
/root/test/test2.txt -> /opt/test/test2.txt
/root/test/test3.txt -> /opt/test/test3.txt

```

sftp 是安全文件传输协议，提供一种安全的加密方法，sftp 是 SSH 的一部分，SFTPClient 类实现了 sftp 客户端，通过已建立的 SSH 通道传输文件，与其他的操作，如下：

方法	描述
sftp.getcwd()	返回当前工作目录
sftp.chdir(path)	改变工作目录
sftp.chmod(path, mode)	修改权限
sftp.chown(path, uid, gid)	设置属主属组
sftp.close()	关闭 sftp
sftp.file(filename, mode='r', bufsize=-1)	读取文件
sftp.from_transport(s)	创建 SFTP 客户端通道
sftp.listdir(path='.')	列出目录，返回一个列表
sftp.listdir_attr(path='.')	列出目录，返回一个 SFTPAttributes 列表
sftp.mkdir(path, mode=511)	创建目录
sftp.normalize(path)	返回规范化 path

方法	描述
<code>sftp.open(filename, mode='r', bufsize=-1)</code>	在远程服务器打开文件
<code>sftp.put(localpath, remotepath, callback=None)</code>	localpath 文件上传到远程服务器 remotepath
<code>sftp.get(remotepath, localpath, callback=None)</code>	从远程服务器 remotepath 拉文件到本地 localpath
<code>sftp.readlink(path)</code>	返回一个符号链接目标
<code>sftp.remove(path)</code>	删除文件
<code>sftp.rename(oldpath, newpath)</code>	重命名文件或目录
<code>sftp.rmdir(path)</code>	删除目录
<code>sftp.stat(path)</code>	返回远程服务器文件信息（返回一个对象的属性）
<code>sftp.truncate(path, size)</code>	截取文件大小
<code>sftp.symlink(source, dest)</code>	创建一个软链接（快捷方式）
<code>sftp.unlink(path)</code>	删除软链接

14.1.5 从远程服务器下载文件

```

import os, sys
import paramiko

hostname = '192.168.0.11'
port = 22
username = 'root'
password = '123456'
local_path = '/root/test.txt'
remote_path = '/opt/test.txt'
try:
    s = paramiko.Transport((hostname, port))
    s.connect(username=username, password=password)
    sftp = paramiko.SFTPClient.from_transport(s)
except Exception as e:
    print(e)
    sys.exit(1)

try:
    # 判断远程服务器是否有这个文件
    sftp.file(remote_path)

```

```

# 使用 get() 方法从远程服务器拉取文件
sftp.get(remote_path, local_path)
except IOError as e:
    print(remote_path + " 远程服务器文件不存在!")
    sys.exit(1)
finally:
    s.close()

# 测试是否下载成功
if os.path.isfile(local_path):
    print("下载成功.")
else:
    print("下载失败!")

# 运行结果
下载成功.

```

14.2 pexpect

pexpect 模块是一个用来启动子程序，并使用正则表达式对程序输出做出特定响应，以此实现与其自动交互。暂不支持 Windows 下的 Python 环境执行。

这里主要讲解 run() 函数和 spawn() 类，能完成自动交互，下面简单了解下它们使用。

官方文档：<http://pexpect.readthedocs.io/en/stable/>

pip 安装：

```
# pip3 install pexpect
```

run() 函数：

run() 函数用来运行 bash 命令，类似于 os 模块中的 system() 函数。

参数：run(command, timeout=-1, withexitstatus=False, events=None, extra_args=None, logfile=None, cwd=None, env=None)

例 1：执行 ls 命令

```
>>> import pexpect
>>> pexpect.run("ls")
```

例 2：获得命令状态返回值

```
>>> command_output, exitstatus = pexpect.run("ls", withexitstatus=1)
```

command_output 是运行结果，exitstatus 是退出状态值。

spawn() 类：

spawn() 是 pexpect 模块主要的类，实现启动子程序，使用 pty.fork() 生成子进程，并调用 exec() 系列函数执行命令。

参数：spawn(command, args=[], timeout=30, maxread=2000, searchwindowsize=None, logfile=None, cwd=None, env=None)

spawn() 类几个常用方法：

方法	描述
<code>expect(pattern, timeout=-1, searchwindowsize=None)</code>	匹配正则表达式， <code>pattern</code> 可以是正则表达式。
<code>send(s)</code>	给子进程发送一个字符串
<code>sendline(s='')</code>	就像 <code>send()</code> ，但添加了一个换行符 (<code>os.lineseq</code>)
<code>sendcontrol(char)</code>	发送一个控制符，比如 <code>ctrl-c</code> 、 <code>ctrl-d</code>

示例：访问 ftp 自动交互

ftp 是一个文件传输协议，用于数据共享，在企业中广泛使用。

为了测试，先装下 vsftpd 软件提供 ftp 服务，与 ftp 命令行工具，看下在命令行是操作的。

```
[root@localhost ~]# yum install vsftpd ftp -y
[root@localhost ~]# systemctl start vsftpd
[root@localhost ~]# ftp 127.0.0.1
Connected to 127.0.0.1 (127.0.0.1).
220 (vsFTPd 3.0.2)
Name (127.0.0.1:root): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
227 Entering Passive Mode (127,0,0,1,221,129).
150 Here comes the directory listing.
drwxr-xr-x  2 0      0          4096 Jun 29 04:22 pub
226 Directory send OK.
ftp> bye
221 Goodbye.
```

默认有一个匿名账户 (anonymous)，密码为空。

默认 ftp 根目录是 `/var/ftp/pub`。

知道了 ftp 登录大致流程，现在写一个 Python 程序使用 `pexpect` 模块自动登录并列出当前目录：

```
import pexpect
child = pexpect.spawn('ftp 127.0.0.1')
child.expect('Name .*: ')
child.sendline('anonymous')
child.expect('Password:')
child.sendline('')
child.expect('ftp> ')
child.sendline('ls')
child.expect('ftp> ')
child.sendline('bye')
```

```
result = child.before
for i in result.decode('utf8').split("\r\n"):
    print(i)

# 运行结果
ls
227 Entering Passive Mode (127,0,0,1,172,109).
150 Here comes the directory listing.
drwxr-xr-x    2 0          0          4096 Jun 29 04:22 pub
226 Directory send OK.
```

Pexpect 有两个重要的方法：expect() 和 send()，该 expect() 方法是等待子应用程序返回给定的字符串。在里面写的字符串是正则表达式，因此可以匹配更复杂的模式。该 send() 方法将一个字符串写入子应用程序。

before 属性将被设置子应用程序打印的文本。返回的是一个 bytes 对象，这里解码成字符串。

小结：手动输入时，是来自键盘的标准输入，而 pexpect 是先匹配到关键字，再向子进程发送字符串。

总结：通过对 Python 下 paramiko 和 pexpect 模块使用，它们各有自己擅长的一面。

paramiko：方便嵌套系统平台中，擅长远程执行命令，文件传输。

pexpect：擅长自动交互，比如 ssh、ftp、telnet。

-- END --

联系作者	李振良（阿良），微信：k8init
官方网站	http://www.aliangedu.cn



阿良个人微信



DevOps技术栈公众号