Python 笔记说明
本文档是本人在学习：
https://www.runoob.com/python3/python3-tutorial.html
https://www.liaoxuefeng.com/wiki/1016959663602400
过程中的记录。
文档中 70%左右的内容都根据以上两个网站，文档中大部分代码都在原有课程基础做过大幅度改进。也存在很多个人原创小案例。
文档内容过多，可能存在瑕疵，欢迎反馈。作者 QQ：604049322 微信：15074804724
编写笔记的过程很累，但能够带来驱动化学习，基本上学完的内容很难忘记，即使忘记也能快速根据自己写的文档回忆起来。
写笔记文档主要是起到了，学过的内容不用害怕忘记的效果。

1. Python 环境

1.1. Python 解释器

当我们编写 Python 代码时，我们得到的是一个包含 Python 代码的以.py 为扩展名的文本文件。要运行代码，就需要 Python 解释器去执行.py 文件。

由于整个 Python 语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写 Python 解释器来执行 Python 代码（当然难度很大）。事实上，确实存在多种 Python 解释器。

1.1.1. CPython

一个官方版本的解释器：CPython。这个解释器是用 C 语言开发的，所以叫 CPython。在命令行下运行 python 就是启动 CPython 解释器。

CPython 是使用最广的 Python 解释器。

1.1.2. IPython

IPython 是基于 CPython 之上的一个交互式解释器，也就是说，IPython 只是在交互方式上有所增强，但是执行 Python 代码的功能和 CPython 是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了 IE。

CPython 用>>>作为提示符，而 IPython 用 In [序号]:作为提示符。

1.1.3. PyPy

PyPy 是另一个 Python 解释器，它的目标是执行速度。PyPy 采用 JIT 技术，对 Python 代码进行动态编译（注意不是解释），所以可以显著提高 Python 代码的执行速度。

绝大部分 Python 代码都可以在 PyPy 下运行，但是 PyPy 和 CPython 有一些是不同的，这就导致相同的 Python 代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到 PyPy 下执行，就需要了解 PyPy 和 CPython 的不同点。

1.1.4. Jython

Jython 是运行在 Java 平台上的 Python 解释器，可以直接把 Python 代码编译成 Java 字节码执行。

1.1.5. IronPython

IronPython 和 Jython 类似，只不过 IronPython 是运行在微软.Net 平台上的 Python 解释器，可以直接把 Python 代码编译成.Net 的字节码。

1.1.6. 说明

Python 的解释器很多，但使用最广泛的还是 CPython。如果要和 Java 或.Net 平台交互，最好的办法不是用 Jython 或 IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

1.2. Linux 安装 Python3

安装依赖环境

```
yum install -y make gcc
```

```
yum install -y zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel readline-devel tk-devel gdbm-devel db4-devel libpcap-devel xz-devel libffi-devel
```

python 安装之前需要一些必要的模块，如 openssl,readline 等。

如果没有这些模块后来使用会出现一些问题，比如没有 openssl 则不支持 ssl 相关的功能，并且 pip3 在安装模块的时候会直接报错；没有 readline 则 python 交互式界面删除键和方向键都无法正常使用，至于需要什么模块在 make 完之后 python 会给出提示，通过提示进行安装即可装全，下面是需要提前预装的依赖：

```
yum -y install zlib zlib-devel
yum -y install bzip2 bzip2-devel
yum -y install ncurses ncurses-devel
yum -y install readline readline-devel
yum -y install openssl openssl-devel
yum -y install openssl-static
yum -y install xz lzma xz-devel
yum -y install sqlite sqlite-devel
yum -y install gdbm gdbm-devel
yum -y install tk tk-devel
yum -y install libffi libffi-devel
```

下载 Python3 安装包

```
wget https://www.python.org/ftp/python/3.7.4/Python-3.7.4.tgz
```

可以在 https://www.python.org/ftp/python/选择要安装的版本

解压安装包

```
tar -zxvf Python-3.7.4.tgz
```

或

```
tar -xvJf  Python-3.7.4.tar.xz
```

进入解压后的目录，编译安装（编译安装前需要安装编译器 **yum install gcc**）个人习惯安装在 **/usr/local/python3**（具体安装位置看个人喜好）

```
cd Python-3.7.4
```

```
mkdir -p /usr/local/python3
```

```
./configure --prefix=/usr/local/python3
```

执行完 configure 命令后，configure 命令执行完之后，会生成一个 Makefile 文件，这个 Makefile 主要是被下一步的 make 命令所使用（ Linux 需要按照 Makefile 所指定的顺序来构建 (build) 程序组件）。

```
make&make install
```

&表示同时执行 2 步， make 实际就是编译源代码，并生成执行文件。make install 实际上是把生成的执行文件拷贝到之前 configure 命令指定的目录/usr/local/python3 下。

到这里安装已经结束，下面是配置环境：

建立 python3 的软链接

```
ln -s /usr/local/python3/bin/python3 /usr/bin/python3
```

```
ln -s /usr/local/python3/bin/pip3 /usr/bin/pip3
```

检查 Python3 是否正常可用：

```
python3 -V
```

额外配置：

./configure --prefix=/usr/local/python3 --enable-shared CFLAGS=-fPIC

#加上--enable-shared 和-fPIC，可以将 python3 的动态链接库编译出来，默认情况编译完 lib 下面只有 python3.xm.a 这样的文件，

python 本身可以正常使用，但是如果编译第三方库需要 python 接口的比如 caffe 等，则会报错；所以这里建议按照以下方式配置，另外如果 openssl 不使用系统 yum 安装的，而是使用自己编译的比较新的版本可以使用 --with-openssl=/usr/local/openssl 这种方式指定，后面目录为 openssl 实际安装的目录，另外编译完还要将 openssl 的 lib 目录加入 ld 运行时目录中即可。

1.3. Linux 环境变量配置

将 xxx 加入 PATH

```
vim /etc/profile
```

然后在文件末尾添加

```
export PATH=$PATH:xxx
```

按 ESC，输入:wq 回车退出。

修改完后，还需要让这个环境变量在配置信息中生效，执行命令：

```
source /etc/profile
```

可以让 profile 文件立即生效。

或修改用户环境变量

vi ~/.bashrc

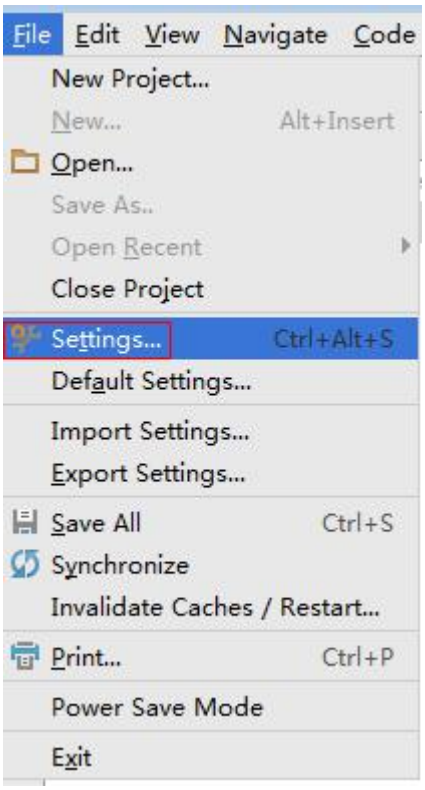
1.4. Pycharm

1.4.1. Python 中文编码代码模板

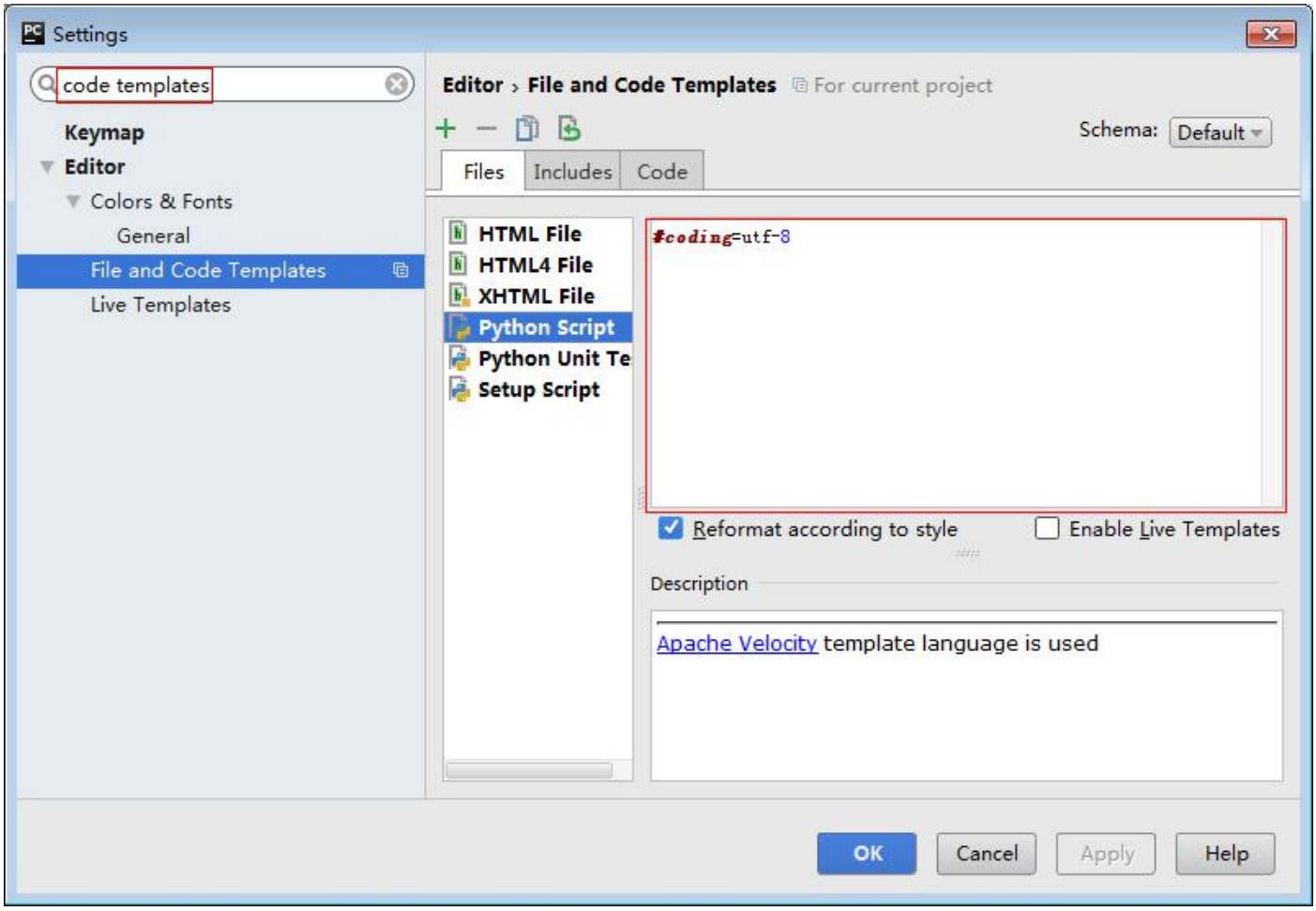
只要在文件开头加入#coding=utf-8 就可以了

由于经常要使用中文，我们完全可以把编码申明写成模板，这样每次创建文件时，都自动有标示

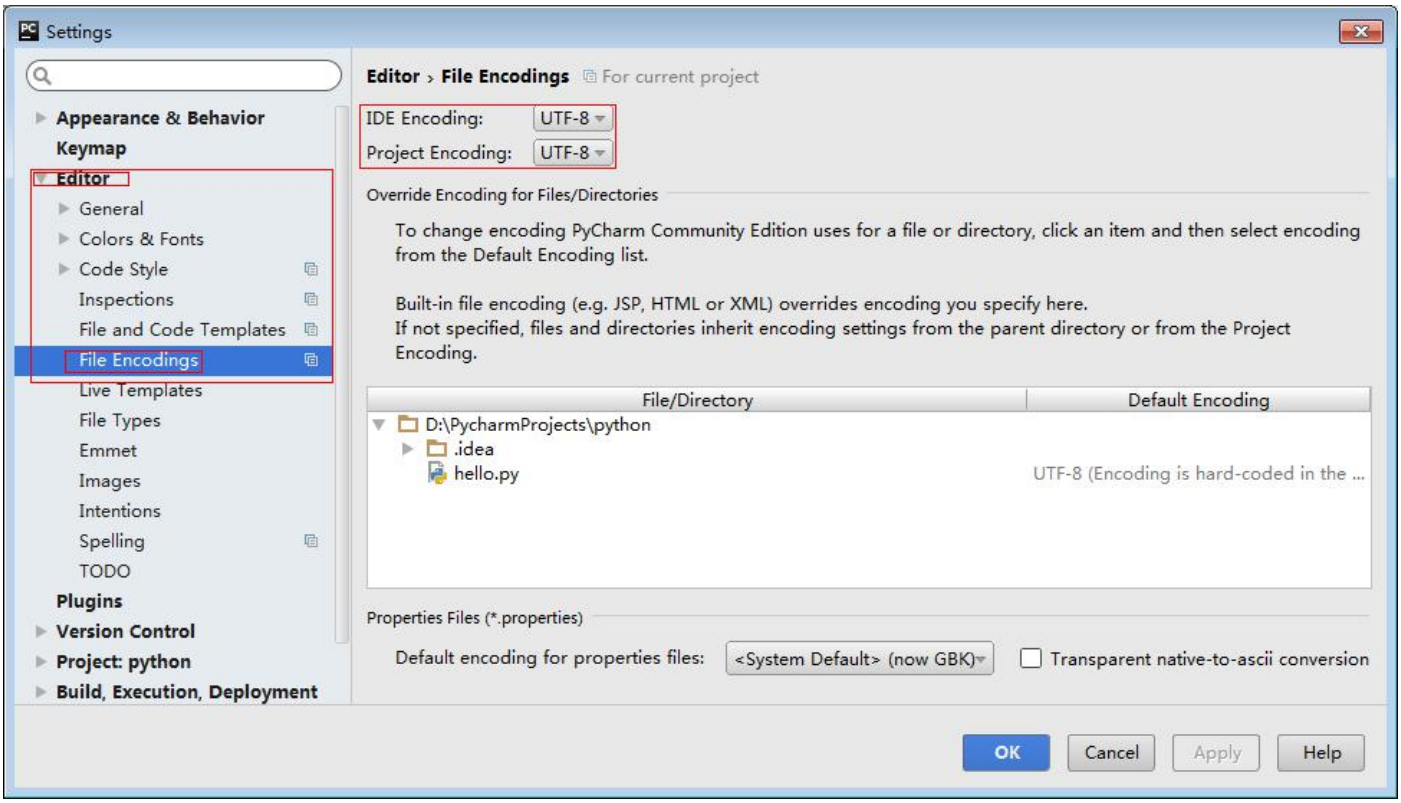
先点击 file->settings



搜索 code templates,在右边红框中输入模板内容



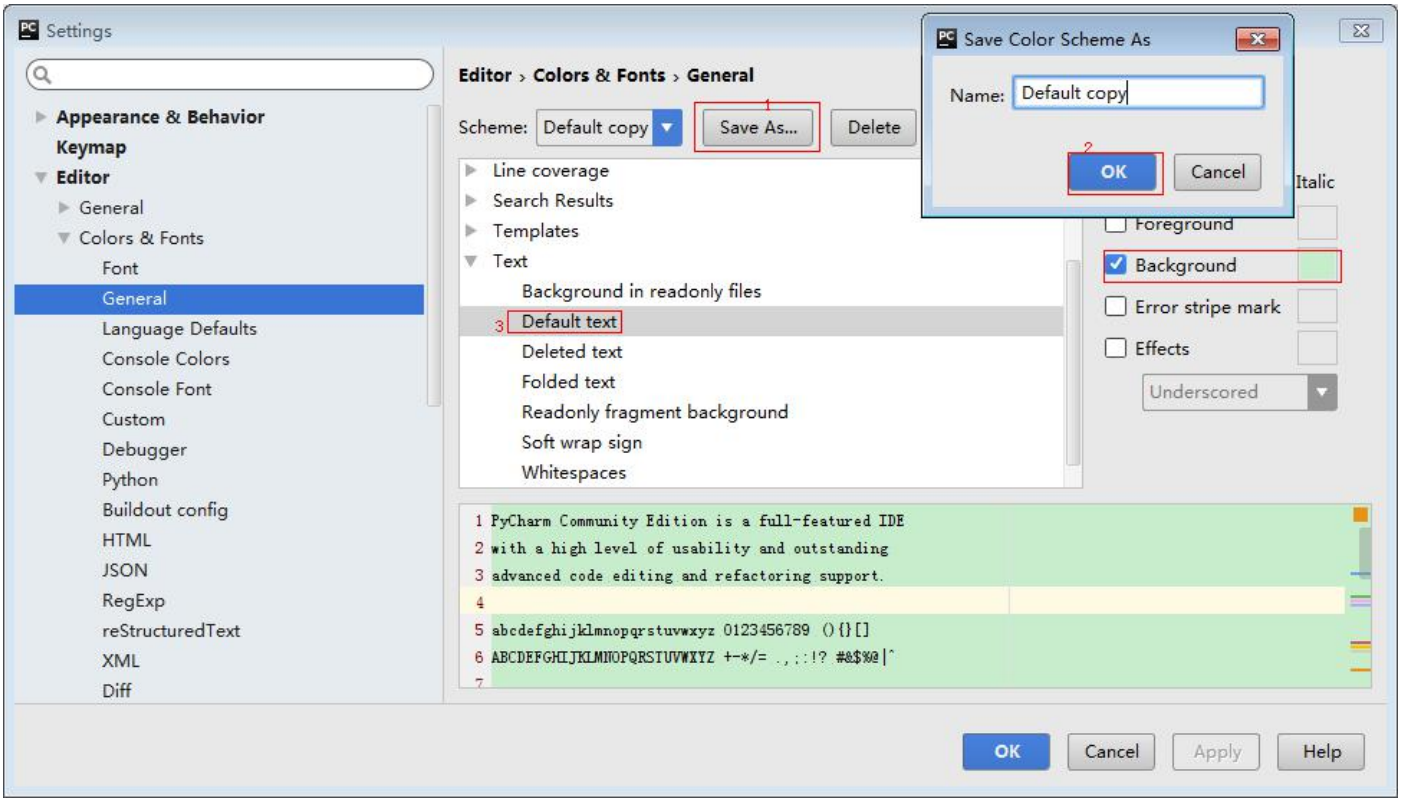
另外把 ide 的默认编码改成 utf-8

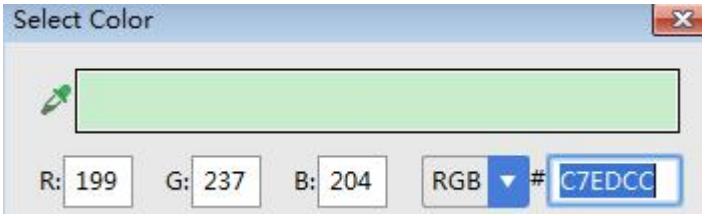


1.4.2. 修改 pycharm 背景色

在 setting->general 中点击 save as 保存一个可编辑的模板，然后在给 defalut text 勾选 backgroud 点击右侧颜色块即

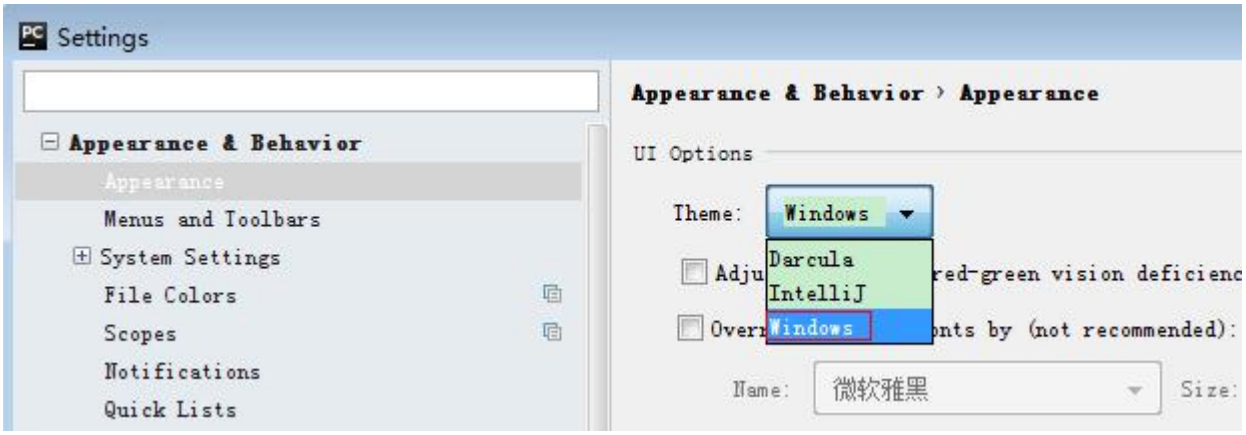
可修改背景色



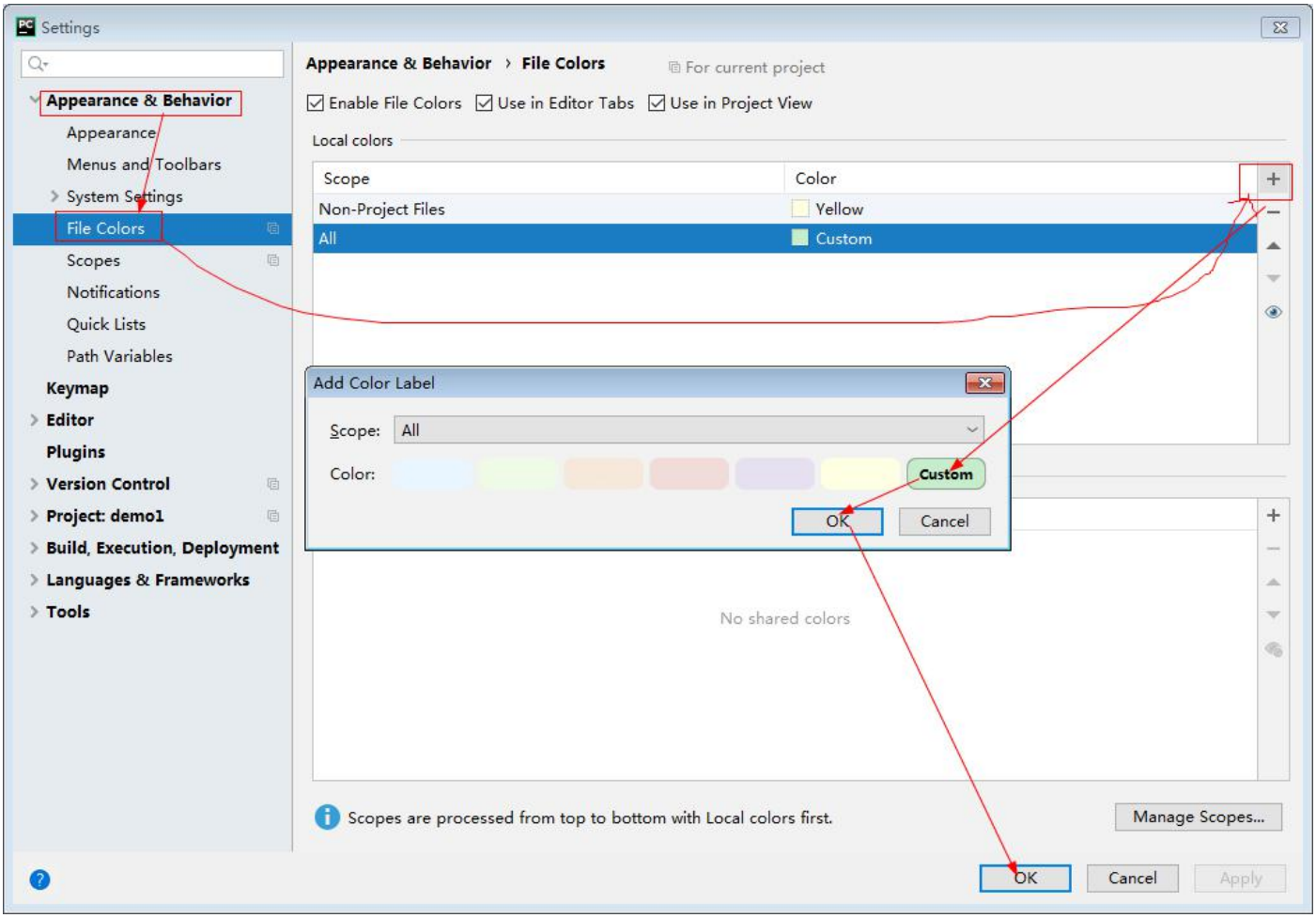


将颜色值修改为 C7EDCC 的护眼色

修改主题为 windows,则界面颜色根据 windows 的颜色设置发生变化

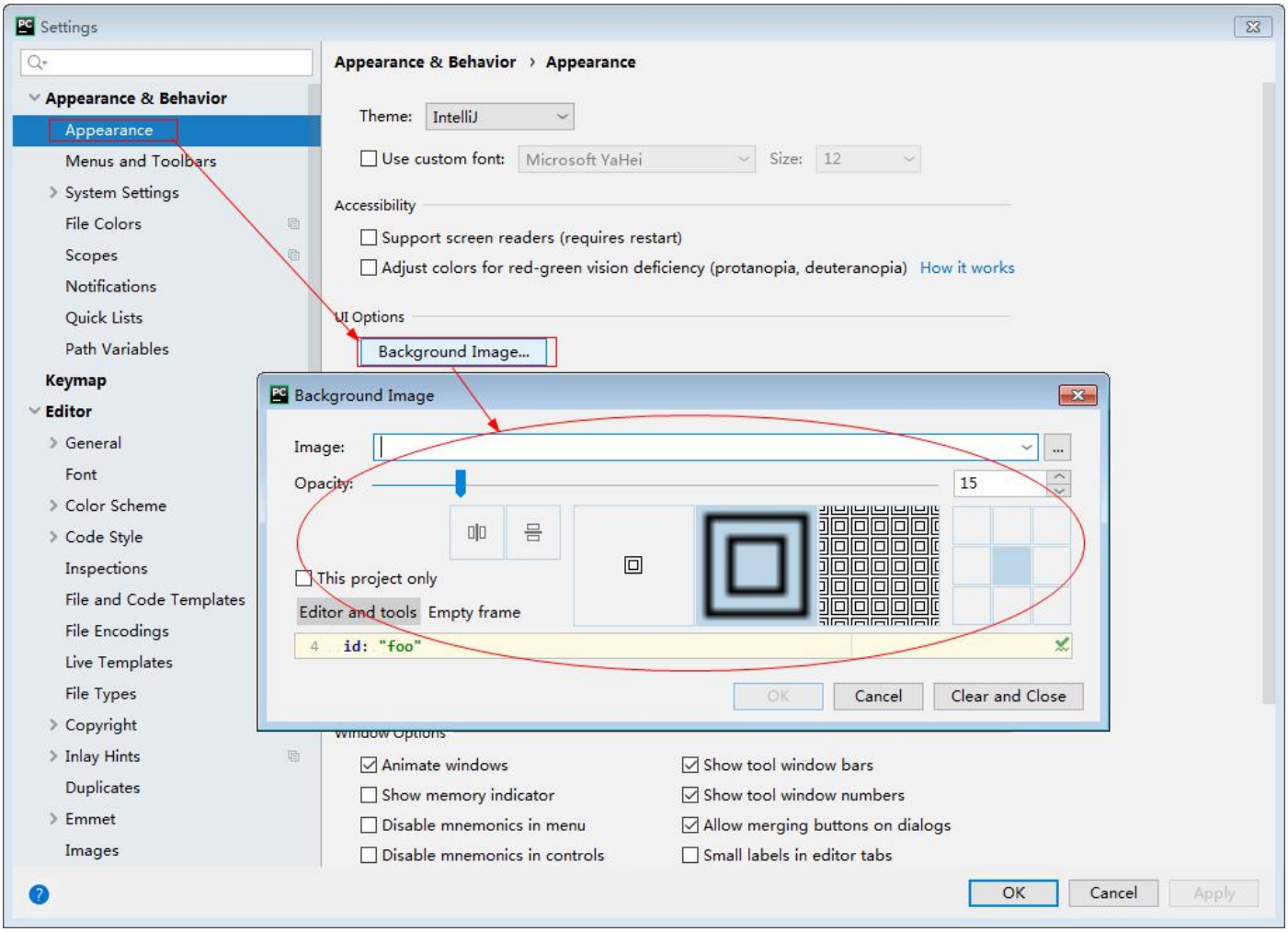


File | Settings | Appearance & Behavior | File Colors

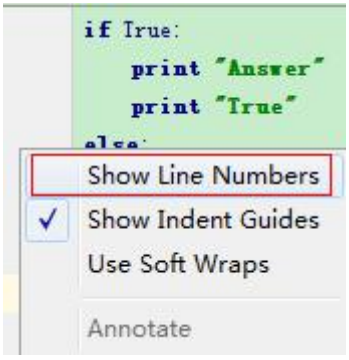


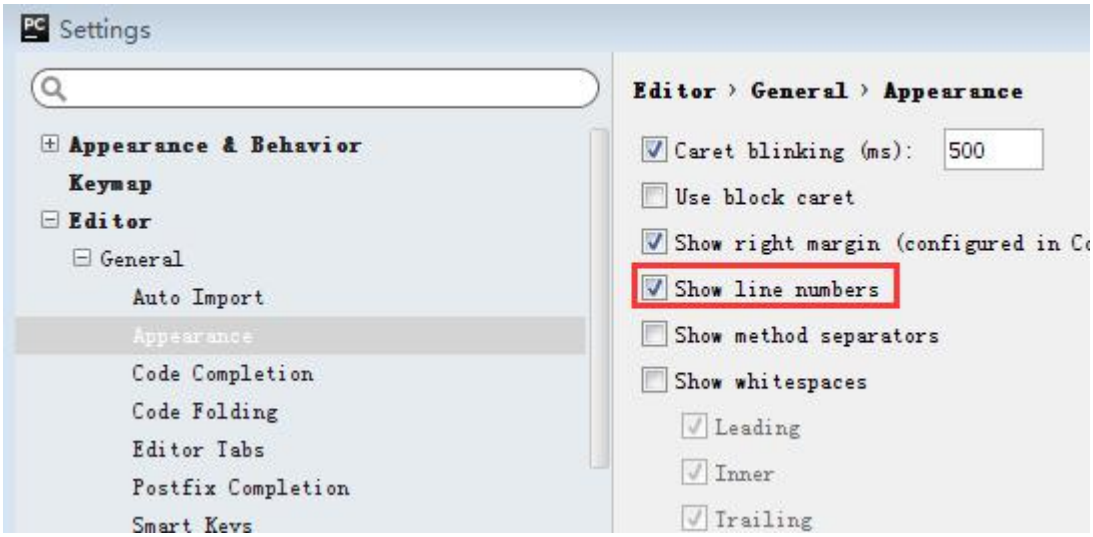
可修改左侧项目背景。

图片背景：


Settings
Appearance & Behavior
Appearance
Menus and Toolbars
System Settings
File Colors
Scopes
Notifications
Quick Lists
Path Variables
Keymap
Editor
General
Font
Color Scheme
Code Style
Inspections
File and Code Templates
File Encodings
Live Templates
File Types
Copyright
Inlay Hints
Duplicates
Emmet
Images
Appearance & Behavior › Appearance
Theme: IntelliJ
Use custom font: Microsoft YaHei Size: 12
Accessibility
Support screen readers (requires restart)
Adjust colors for red-green vision deficiency (protanopia, deuteranopia) How it works
UI Options
Background Image...
Background Image
Image:
Opacity:
15
This project only
Editor and tools
Empty frame
4 id: "foo"
OK
Cancel
Clear and Close
Window Options
Animate windows
Show memory indicator
Disable mnemonics in menu
Disable mnemonics in controls
Show tool window bars
Show tool window numbers
Allow merging buttons on dialogs
Small labels in editor tabs
OK
Cancel
Apply

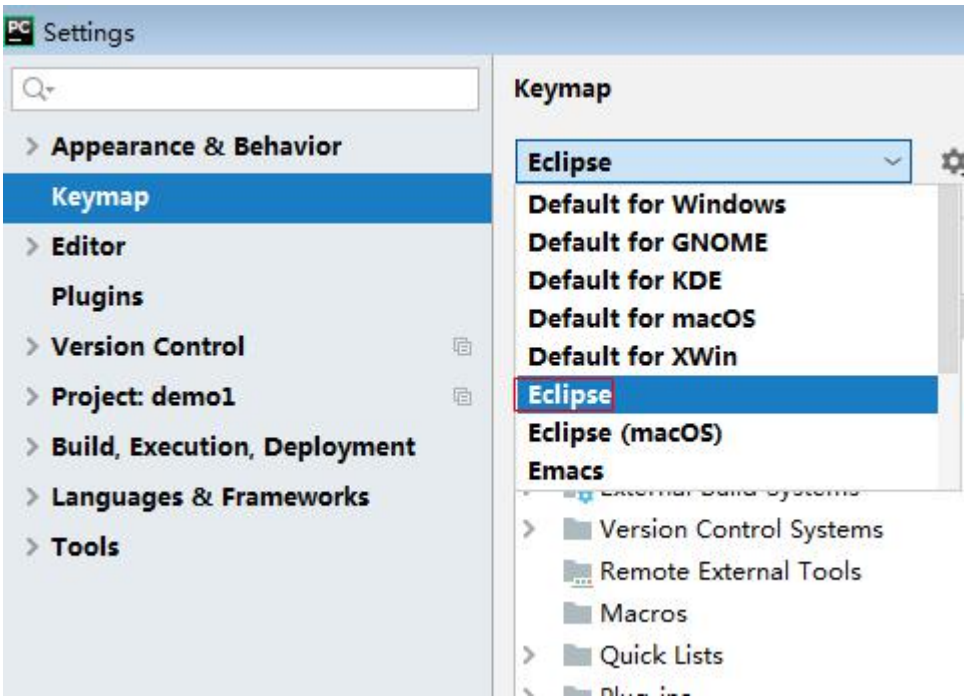
1.4.3. 显示行号


if True:
print "Answer"
print "True"
Show Line Numbers
Show Indent Guides
Use Soft Wraps
Annotate



1.4.4. Eclipse 快捷键

Pycharm 设置 Eclipse 快捷键



Ctrl+1 快速修复
Ctrl+D: 删除当前行
Ctrl+Alt+↓ 复制当前行到下一行(复制增加)
Ctrl+Alt+↑ 复制当前行到上一行(复制增加)
Alt+↓ 当前行和下面一行交互位置(特别实用,可以省去先剪切,再粘贴了)
Alt+↑ 当前行和上面一行交互位置(同上)
Alt+← 前一个编辑的页面
Alt+→ 下一个编辑的页面(当然是针对上面那条来说了)
Alt+Enter 显示当前选择资源(工程,or 文件 or 文件)的属性
Shift+Enter 在当前行的下一行插入空行(这时鼠标可以在当前行的任一位置,不一定是最后)
Shift+Ctrl+Enter 在当前行插入空行(原理同上条)
Ctrl+Q 定位到最后编辑的地方
Ctrl+L 定位在某行 (对于程序超过 100 的人就有福音了)
Ctrl+M 最大化当前的 Edit 或 View (再按则反之)

Ctrl+/ 注释当前行,再按则取消注释
Ctrl+O 快速显示 OutLine
Ctrl+T 快速显示当前类的继承结构
Ctrl+W 关闭当前 Editer
Ctrl+K 参照选中的 Word 快速定位到下一个
Ctrl+E 快速显示当前 Editer 的下拉列表(如果当前页面没有显示的用黑体表示)
Ctrl+/(小键盘) 折叠当前类中的所有代码
Ctrl+×(小键盘) 展开当前类中的所有代码
Ctrl+Space 代码助手完成一些代码的插入(但一般和输入法有冲突,可以修改输入法的热键,也可以暂用 Alt+/来代替)
Ctrl+Shift+E 显示管理当前打开的所有的 View 的管理器(可以选择关闭,激活等操作)
Ctrl+J 正向增量查找(按下 Ctrl+J 后,你所输入的每个字母编辑器都提供快速匹配定位到某个单词,如果没有,则在 stutes line 中显示没有找到了,查一个单词时,特别实用,这个功能 Idea 两年前就有了)
Ctrl+Shift+J 反向增量查找(和上条相同,只不过是从后往前查)
Ctrl+Shift+F4 关闭所有打开的 Editer
Ctrl+Shift+X 把当前选中的文本全部变味小写
Ctrl+Shift+Y 把当前选中的文本全部变为小写
Ctrl+Shift+F 格式化当前代码

Ctrl+Shift+P 定位到对于的匹配符(譬如{}) (从前面定位后面时,光标要在匹配符里面,后面到前面,则反之)

注:一般重构的快捷键都是 Alt+Shift 开头的了
Alt+Shift+R 重命名 (是我自己最爱用的一个了,尤其是变量和类的 Rename,比手工方法能节省很多劳动力)
Alt+Shift+M 抽取方法 (这是重构里面最常用的方法之一了,尤其是对一大堆泥团代码有用)
Alt+Shift+C 修改函数结构(比较实用,有 N 个函数调用了这个方法,修改一次搞定)
Alt+Shift+L 抽取本地变量(可以直接把一些魔法数字和字符串抽取成一个变量,尤其是多处调用的时候)
Alt+Shift+F 把 Class 中的 local 变量变为 field 变量 (比较实用的功能)
Alt+Shift+I 合并变量(可能这样说有点不妥 Inline)
Alt+Shift+V 移动函数和变量(不怎么常用)
Alt+Shift+Z 重构的后悔药(Undo)

编辑
作用域 功能 快捷键
全局 查找并替换 Ctrl+F
文本编辑器 查找上一个 Ctrl+Shift+K
文本编辑器 查找下一个 Ctrl+K
全局 撤销 Ctrl+Z
全局 复制 Ctrl+C
全局 恢复上一个选择 Alt+Shift+↓
全局 剪切 Ctrl+X
全局 快速修正 Ctrl1+1
全局 内容辅助 Alt+/
全局 全部选中 Ctrl+A
全局 删除 Delete
全局 上下文信息 Alt+?
Alt+Shift+?
Ctrl+Shift+Space
Java 编辑器 显示工具提示描述 F2
Java 编辑器 选择封装元素 Alt+Shift+↑

Java 编辑器 选择上一个元素 Alt+Shift+←
Java 编辑器 选择下一个元素 Alt+Shift+→
文本编辑器 增量查找 Ctrl+J
文本编辑器 增量逆向查找 Ctrl+Shift+J
全局 粘贴 Ctrl+V
全局 重做 Ctrl+Y

查看
作用域 功能 快捷键
全局 放大 Ctrl+=
全局 缩小 Ctrl+-

窗口
作用域 功能 快捷键
全局 激活编辑器 F12
全局 切换编辑器 Ctrl+Shift+W
全局 上一个编辑器 Ctrl+Shift+F6
全局 上一个视图 Ctrl+Shift+F7
全局 上一个透视图 Ctrl+Shift+F8
全局 下一个编辑器 Ctrl+F6
全局 下一个视图 Ctrl+F7
全局 下一个透视图 Ctrl+F8
文本编辑器 显示标尺上下文菜单 Ctrl+W
全局 显示视图菜单 Ctrl+F10
全局 显示系统菜单 Alt+-

导航
作用域 功能 快捷键
Java 编辑器 打开结构 Ctrl+F3
全局 打开类型 Ctrl+Shift+T
全局 打开类型层次结构 F4
全局 打开声明 F3
全局 打开外部 javadoc Shift+F2
全局 打开资源 Ctrl+Shift+R
全局 后退历史记录 Alt+←
全局 前进历史记录 Alt+→
全局 上一个 Ctrl+,
全局 下一个 Ctrl+.
Java 编辑器 显示大纲 Ctrl+O
全局 在层次结构中打开类型 Ctrl+Shift+H
全局 转至匹配的括号 Ctrl+Shift+P
全局 转至上一个编辑位置 Ctrl+Q
Java 编辑器 转至上一个成员 Ctrl+Shift+↑
Java 编辑器 转至下一个成员 Ctrl+Shift+↓
文本编辑器 转至行 Ctrl+L

搜索
作用域 功能 快捷键
全局 出现在文件中 Ctrl+Shift+U
全局 打开搜索对话框 Ctrl+H
全局 工作区中的声明 Ctrl+G
全局 工作区中的引用 Ctrl+Shift+G

文本编辑
作用域 功能 快捷键
文本编辑器 改写切换 Insert
文本编辑器 上滚行 Ctrl+↑
文本编辑器 下滚行 Ctrl+↓

文件
作用域 功能 快捷键
全局 保存 Ctrl+X
Ctrl+S
全局 打印 Ctrl+P
全局 关闭 Ctrl+F4
全局 全部保存 Ctrl+Shift+S
全局 全部关闭 Ctrl+Shift+F4
全局 属性 Alt+Enter
全局 新建 Ctrl+N

项目
作用域 功能 快捷键
全局 全部构建 Ctrl+B
源代码
作用域 功能 快捷键
Java 编辑器 格式化 Ctrl+Shift+F
Java 编辑器 取消注释 Ctrl+\
Java 编辑器 注释 Ctrl+/
Java 编辑器 添加导入 Ctrl+Shift+M
Java 编辑器 组织导入 Ctrl+Shift+O
Java 编辑器 使用 try/catch 块来包围 未设置，太常用了，所以在这里列出,建议自己设置。
也可以使用 Ctrl+1 自动修正。
运行
作用域 功能 快捷键
全局 单步返回 F7
全局 单步跳过 F6
全局 单步跳入 F5
全局 单步跳入选择 Ctrl+F5
全局 调试上次启动 F11
全局 继续 F8
全局 使用过滤器单步执行 Shift+F5
全局 添加/去除断点 Ctrl+Shift+B
全局 显示 Ctrl+D

全局 运行上次启动 Ctrl+F11
全局 运行至行 Ctrl+R
全局 执行 Ctrl+U

重构
作用域 功能 快捷键
全局 撤销重构 Alt+Shift+Z
全局 抽取方法 Alt+Shift+M
全局 抽取局部变量 Alt+Shift+L
全局 内联 Alt+Shift+I
全局 移动 Alt+Shift+V
全局 重命名 Alt+Shift+R
全局 重做 Alt+Shift+Y

2. Python 基础语法

2.1. Python 标识符

标识符第一个字符必须是字母表中字母或下划线 _ ，**不能以数字开头**。
标识符的其他的部分由字母、数字和下划线组成。
标识符对大小写敏感。
在 Python 3 中，可以用中文作为变量名，非 ASCII 标识符也是允许的了。

以下划线开头的标识符是有特殊意义的：
以**单下划线开头 _foo 的代表不能直接访问的类属性**，需通过类提供的接口进行访问，不能用 from xxx import * 而导入；
以**双下划线开头的 __foo 代表类的私有成员**；以**双下划线开头和结尾**的 __foo__ 代表 Python 里特殊方法专用的标识，如 __init__() 代表类的构造函数。

Python 可以同一行显示多条语句，方法是用分号 ；分开。

2.2. Python 关键字

又称保留字，Python 的标准库提供了一个 keyword 模块，可以输出当前版本的所有关键字：

```
import keyword
print(keyword.kwlist)
```

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

2.3. 行和缩进

学习 Python 与其他语言最大的区别就是，Python 的代码块不使用大括号 {} 来控制类，函数以及其他逻辑判断。

python 最具特色的就是用缩进对齐来写模块。
缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。

2.4. 多行语句

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

```
total = item_one + \
        item_two + \
        item_three
```

在 [], {}, 或 () 中的多行语句，不需要使用反斜杠(\)，例如：

```
total = ['item_one', 'item_two', 'item_three',
        'item_four', 'item_five']
```

2.5. Python 引号与注释

Python 可以使用引号(')、双引号(")、三引号(''' 或 """) 来表示字符串，引号的开始与结束必须的相同类型的。
其中三引号可以由多行组成，编写多行文本的快捷语法，还可以当做注释。
python 中多行注释使用三个单引号(''')或三个双引号(""")。
#表示单行注释，单行注释可以单独一行也可以在语句或表达式行末：

```
word = 'word'
sentence = "这是一个句子。"
paragraph = """这是一个段落。
包含了多个语句"""

'''
这是多行注释，使用单引号。
这是多行注释，使用单引号。
这是多行注释，使用单引号。
'''

"""
这是多行注释，使用双引号。
这是多行注释，使用双引号。
这是多行注释，使用双引号。
"""

# 第一个注释
print("Hello, Python!")  # 第二个注释
```

2.6. Python 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。
空行与代码缩进不同，空行并不是 Python 语法的一部分。书写时不插入空行，Python 解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

2.7. Input 读取用户输入

Python 提供了 input() 内置函数从标准输入读入一行文本，默认的标准输入是键盘。

input 可以接收一个 Python 表达式作为输入，并将运算结果返回。
实例：

```
# coding=utf-8

str = input("请输入：")
print("你输入的内容是: ", str)
```

结果：

```
请输入：hello python
你输入的内容是:  hello python
```

2.8. 变量赋值与删除

Python 中的变量赋值不需要类型声明。
每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。
等号（=）用来给变量赋值。
等号（=）运算符左边是一个变量名,等号（=）运算符右边是存储在变量中的值。例如：

```
# coding=utf-8

counter = 100  # 赋值整型变量
miles = 1000.0  # 浮点型
name = "John"  # 字符串

print(counter)
print(miles)
print(name)
# 也可以一次性打印多个变量
print(counter, miles, name)
# del 可删除对象引用
del counter, miles, name

# Python 允许你同时为多个变量赋值。以下实例，创建一个整型对象，值为 1，三个变量被分配到相同的内存空间上
a = b = c = 1
# 您也可以为多个对象指定多个变量
a, b, c = 1, 2, "john"
```

在 Python 中，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量。
这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。

2.9. pass 语句

Python pass 是空语句，是为了保持程序结构的完整性。
pass 不做任何事情，一般用做占位语句。

```
for letter in 'Python':
    pass
```

2.10. import 与 from...import

在 python 用 `import` 或者 `from...import` 来导入相应的模块。

将整个模块(somemodule)导入，格式为： `import somemodule`

从某个模块中导入某个函数,格式为： `from somemodule import somefunction`

从某个模块中导入多个函数,格式为： `from somemodule import firstfunc, secondfunc, thirdfunc`

将某个模块中的全部函数导入，格式为： `from somemodule import *`

2.11. Python3 标准库概览

2.11.1. 文件通配符

glob 模块提供了一个函数用于从目录通配符搜索中生成文件列表:

```
>>> import glob

>>> glob.glob('*.py')

['primes.py', 'random.py', 'quote.py']
```

2.11.2. 命令行参数

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

```
>>> import sys

>>> print(sys.argv)

['demo.py', 'one', 'two', 'three']
```

错误输出重定向和程序终止
sys 还有 stdin，stdout 和 stderr 属性，即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息。

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

Warning, log file not found starting a new one
```

大多脚本的定向终止都使用 "sys.exit()"。

```
import sys
sys.stderr = open("error.log", "a")
```

可实现错误输出重定向。

2.11.3. 访问互联网

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 urllib.request

```
from urllib.request import urlopen
for line in urlopen('http://www.baidu.com/'):
    line = line.decode('utf-8')
    print(line)
```

2.11.4. 日期和时间

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

```
from datetime import date
now = date.today()
print(now)
print(now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B."))

birthday = date(2008, 10, 1)
age = now - birthday
print(age.days)
```

结果:

```
2019-10-01
10-01-19. 01 Oct 2019 is a Tuesday on the 01 day of October.
4017
```

2.12. Python 运算符

2.12.1. Python 算术运算符

以下假设变量： a=10，b=20：

运算符	描述	实例
+	加:两个对象相加	a + b 输出结果 30
-	减:得到负数或是一个数减去另一个数	a - b 输出结果 -10
*	乘:两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除:x 除以 y	b / a 输出结果 2
%	取模:返回除法的余数	b % a 输出结果 0
**	幂:返回 x 的 y 次幂	a**b 为 10 的 20 次方， 输出结果 100000000000000000000
//	取整除:返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

```
#coding=utf-8

a,b = 21,10
print("a=",a,"b=",b)
print("a + b 的值为：", a + b)
print("a - b 的值为：",a - b)
print("a * b 的值为：", a * b)
print("a / b 的值为：", a / b)
print("a % b 的值为：", a % b)
print("2**3 的值为：", 2**3)
print("10//2 的值为：", 10//2)
```

结果:

```
a= 21 b= 10
a + b 的值为： 31
a - b 的值为： 11
a * b 的值为： 210
a / b 的值为： 2.1
a % b 的值为： 1
2**3 的值为： 8
10//2 的值为： 5
```

2.12.2. Python 比较运算符

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true.
>	大于 - 返回 x 是否大于 y	(a > b) 返回 False。

<	小于 - 返回 x 是否小于 y。所有比较运算符返回 1 表示真，返回 0 表示假。这分别与特殊的变量 True 和 False 等价。注意，这些变量名的大写。	(a < b) 返回 true。
>=	大于等于 - 返回 x 是否大于等于 y。	(a >= b) 返回 False。
<=	小于等于 - 返回 x 是否小于等于 y。	(a <= b) 返回 true。

2.12.3. Python 赋值运算符

运算符	描述	实例
=	简单的赋值	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值	c += a 等效于 c = c + a
-=	减法赋值	c -= a 等效于 c = c - a
*=	乘法赋值	c *= a 等效于 c = c * a
/=	除法赋值	c /= a 等效于 c = c / a
%=	取模赋值	c %= a 等效于 c = c % a
**=	幂赋值	c **= a 等效于 c = c ** a
//=	取整除赋值	c //= a 等效于 c = c // a

2.12.4. Python 位运算符

按位运算符是把数字看作二进制来进行计算的。Python 中的按位运算法则如下：

运算符	描述
&	按位与运算符：参与运算的两个值,如果两个相应位都为 1,则该位的结果为 1,否则为 0
|	按位或运算符：只要对应的二个二进位有一个为 1 时，结果位就为 1。
^	按位异或运算符：当两对应的二进位相异时，结果为 1
~	按位取反运算符：对数据的每个二进制位取反,即把 1 变为 0,把 0 变为 1。**~x** 类似于 **-x-1**
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补 0。
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数

& 按位与运算符：参与运算的两个值,如果两个相应位都为 1,则该位的结果为 1,否则为 0
| 按位或运算符：只要对应的二个二进位有一个为 1 时，结果位就为 1。
^ 按位异或运算符：当两对应的二进位相异时，结果为 1 (a ^ b)
~ 按位取反运算符：对数据的每个二进制位取反,即把 1 变为 0,把 0 变为 1
<< 左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补 0。
>> 右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数 a >> 2

下表中变量 a 为 60，b 为 13，二进制格式如下：
a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100

a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011
a<<2= 1111 0000
a>>2= 0000 1111
例：

```
def decInt2Bin(string_num):
    num = int(string_num)
    if (num < 0):
        num = abs(num) ^ 0xFF - 1
    result = bin(num)[2:]
    return result.zfill(8)

a = 60  # 60 = 0011 1100
b = 13  # 13 = 0000 1101
print('dec:a=', a, 'bin:a=', decInt2Bin(a))
print('dec:b=', b, 'bin:b=', decInt2Bin(b))
print('dec:a&b=', a & b, 'bin:a&b=', decInt2Bin(a & b))
print('dec:a|b=', a | b, 'bin:a|b=', decInt2Bin(a | b))
print('dec:a^b=', a ^ b, 'bin:a^b=', decInt2Bin(a ^ b))

print('dec:a<<2=', a << 2, 'bin:a<<2=', decInt2Bin(a << 2))
print('dec:a>>2=', a >> 2, 'bin:a>>2=', decInt2Bin(a >> 2))

print('dec:~a=', ~a, 'bin:~a=', decInt2Bin(~a))  # -61 = 1100 0011
```

结果:

```
dec:a= 60 bin:a= 00111100
dec:b= 13 bin:b= 00001101
dec:a&b= 12 bin:a&b= 00001100
dec:a|b= 61 bin:a|b= 00111101
dec:a^b= 49 bin:a^b= 00110001
dec:a<<2= 240 bin:a<<2= 11110000
dec:a>>2= 15 bin:a>>2= 00001111
dec:~a= -61 bin:~a= 11000011
```

2.12.5. Python 逻辑运算符

x and y　x 和 y 都为 true 时，返回 true，否则返回 false
x or y　x 和 y 都为 false 时，返回 false，否则返回 true
not　not x　如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。

对于数值，python 认为非 0 表示 True，0 表示 False
例：

```
a, b = 10, 20

if (a and b):
    print("1 - 变量 a 和 b 都为 true")
else:
```

```
    print("1 - 变量 a 和 b 有一个不为 true")

if (a or b):
    print("2 - 变量 a 和 b 都为 true，或其中一个变量为 true")
else:
    print("2 - 变量 a 和 b 都不为 true")

# 修改变量 a 的值
a = 0
if (a and b):
    print("3 - 变量 a 和 b 都为 true")
else:
    print("3 - 变量 a 和 b 有一个不为 true")

if (a or b):
    print("4 - 变量 a 和 b 都为 true，或其中一个变量为 true")
else:
    print("4 - 变量 a 和 b 都不为 true")

if not (a and b):
    print("5 - 变量 a 和 b 都为 false，或其中一个变量为 false")
else:
    print("5 - 变量 a 和 b 都为 true")
```

结果：

```
1 - 变量 a 和 b 都为 true
2 - 变量 a 和 b 都为 true，或其中一个变量为 true
3 - 变量 a 和 b 有一个不为 true
4 - 变量 a 和 b 都为 true，或其中一个变量为 true
5 - 变量 a 和 b 都为 false，或其中一个变量为 false
```

2.12.6. Python 成员运算符

除了以上的一些运算符之外，Python 还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	x 在 y 序列中 , 如果 x 在 y 序列中返回 True。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	x 不在 y 序列中 , 如果 x 不在 y 序列中返回 True。

```
a = 10
b = 20
list = [1, 2, 3, 4, 5]

if (a in list):
    print("1 - 变量 a 在给定的列表中 list 中")
```

```
else:
    print("1 - 变量 a 不在给定的列表中 list 中")

if (b not in list):
    print("2 - 变量 b 不在给定的列表中 list 中")
else:
    print("2 - 变量 b 在给定的列表中 list 中")

# 修改变量 a 的值
a = 2
if (a in list):
    print("3 - 变量 a 在给定的列表中 list 中")
else:
    print("3 - 变量 a 不在给定的列表中 list 中")
```

结果：

```
1 - 变量 a 不在给定的列表中 list 中
2 - 变量 b 不在给定的列表中 list 中
3 - 变量 a 在给定的列表中 list 中
```

2.12.7. Python 身份运算符

身份运算符用于比较两个对象的存储单元是否引用自一个对象

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y, 如果 id(x) 等于 id(y) , is 返回结果 1
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y, 如果 id(x) 不等于 id(y). is not 返回结果 1

例：

```
a, b = 20, 20

print("a is b:", a is b)
print("id(a) == id(b):", id(a) == id(b))
b = 30
print("a is b:", a is b)
print("a is not b:", a is not b)
```

结果：

```
a is b: True
id(a) == id(b): True
a is b: False
a is not b: True
```

is 与 == 区别：

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

```
>>>a = [1, 2, 3]
>>> b = a >>> b is a
```

```
True
>>> b == a
True
>>> b = a[:]
>>> b is a
False
>>> b == a
True
```

2.12.8. Python 运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
== !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not and or	逻辑运算符

2.13. Python 条件语句

2.13.1. 基本 if 语句

if 判断条件：
    执行语句……
else：
    执行语句……
例：

```
# coding=utf-8

# 例1：if 基本用法

name = 'luren'
if name == 'python':  # 判断变量否为'python'
    print('welcome boss')  # 并输出欢迎信息
else:
    print(name)
```

结果：luren
上例表示，当 name 等于 python 成立时，打印 welcome boss，否则打印 name 变量的值

2.13.2. 复杂 if 语句

if 判断条件 1:
    执行语句 1……
elif 判断条件 2:
    执行语句 2……
elif 判断条件 3:
    执行语句 3……
else:

执行语句 4……

例：

```
# coding=utf-8

# 例2：elif 用法

num = 5
if num == 3:  # 判断num的值
    print('boss')
elif num == 2:
    print('user')
elif num == 1:
    print('worker')
elif num < 0:  # 值小于零时输出
    print('error')
else:
    print('roadman')  # 条件均不成立时输出
```

结果：roadman

2.13.3. 三元表达式

格式为：

为真时的结果 if 判定条件 else 为假时的结果

例：

```
print(1 if 5 > 3 else 0)
print(1 if 5 < 3 else 0)
```

结果：

1

0

2.14. Python 循环语句

Python for 循环嵌套语法：

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

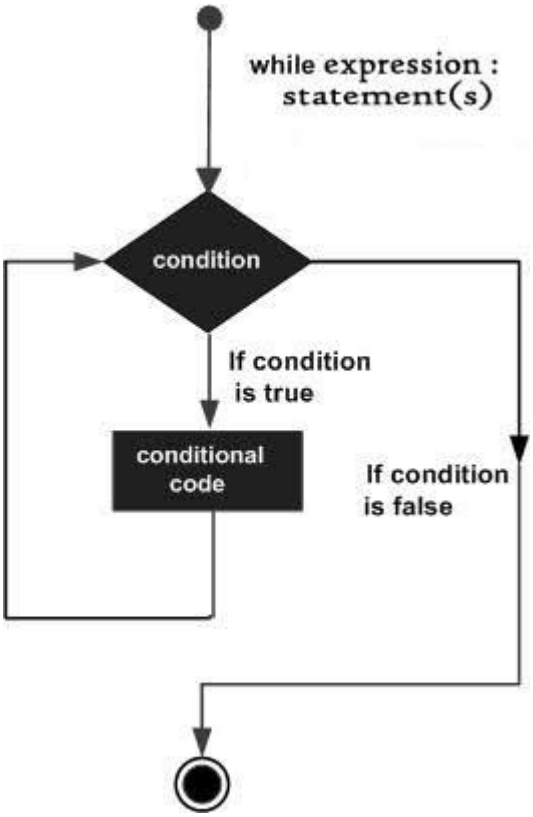
Python while 循环嵌套语法：

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

你可以在循环体内嵌入其他的循环体，如在 while 循环中可以嵌入 for 循环， 反之，你可以在 for 循环中嵌入 while 循环。

2.14.1. while 循环

执行流程图如下：



while 循环就是不停的判断条件是否成立，条件成立则执行循环体，条件不成立时不执行循环体，退出循环

例:

```
count = 0
while (count < 5):
    print('The count is:', count)
    count = count + 1

print("Good bye!")
```

运行结果：

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
Good bye!
```

while 语句时还有另外两个重要的命令 continue,break 来跳过循环,continue 用于跳过该次循环，break 则是用于退出循环，此外"判断条件"还可以是个常值，表示循环必定成立。

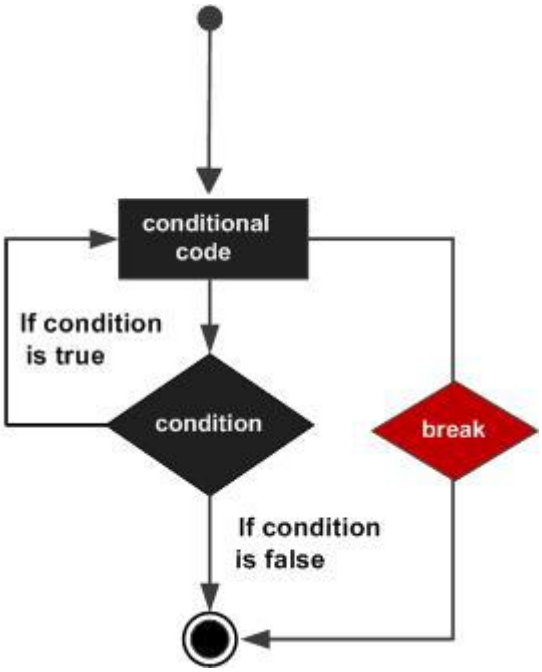
break 语句

Python break 语句，就像在 C 语言中，打破了最小封闭 for 或 while 循环。

break 语句用来终止循环语句，即循环条件没有 False 条件或者序列还没被完全递归完，也会停止执行循环语句。

break 语句用在 while 和 for 循环中。

如果您使用嵌套循环，break 语句将停止执行最深层的循环，并开始执行下一行代码。

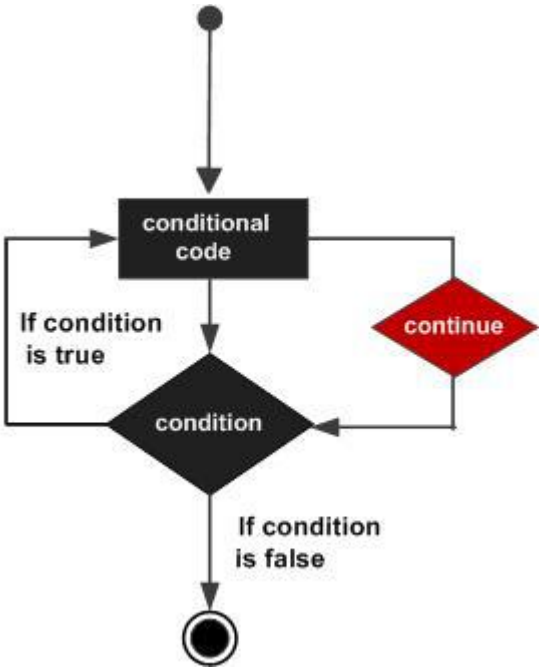


Python continue 语句

Python continue 语句跳出本次循环，而 break 跳出整个循环。

continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。

continue 语句用在 while 和 for 循环中。



具体用法如下：

```
# coding=utf-8

# continue 和 break 用法

i = 1
while i < 5:
    i += 1
    if i % 2 > 0:  # 非双数时跳过输出
        continue
    print(i)  # 输出双数 2、4

i = 1
```

```
while 1:  # 循环条件为 1 必定成立
    print(i)  # 输出 1~10
    i += 1
    if i > 5:  # 当 i 大于 5 时跳出循环
        break
```

结果：

```
2
4
1
2
3
4
5
```

while 循环使用 else 语句

else 的作用是判断循环是否正常结束，如果循环是被 break 中断 else 里的代码块就不再执行

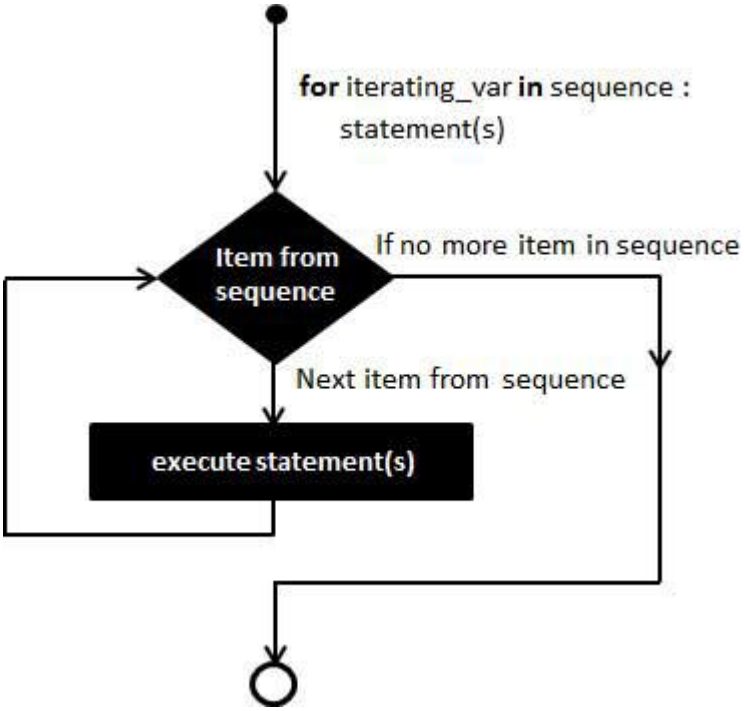
```
count = 0
while count < 5:
    print(count, " is  less than 5")
    count = count + 1
else:
    print(count, " is not less than 5")
```

2.14.2. for 循环

for 循环的语法格式如下：

```
for iterating_var in sequence:
    statements(s)
```

流程图：



for 循环可以遍历任何序列的项目，如一个列表或者一个字符串

例：

```
# coding=utf-8

for letter in 'Python':  # 第一个实例
    print('当前字母 :', letter)

fruits = ['banana', 'apple', 'mango']
for fruit in fruits:  # 第二个实例
    print('当前水果 :', fruit)

print("Good bye!")
```

结果：

```
当前字母 : P
当前字母 : y
当前字母 : t
当前字母 : h
当前字母 : o
当前字母 : n
当前水果 : banana
当前水果 : apple
当前水果 : mango
Good bye!
```

通过索引序列迭代元素

在讲 list 时，我们用到 range 生成 list 序列，现在通过迭代 list 序列获取索引角标

例：

```
# coding=utf-8

fruits = ['banana', 'apple', 'mango']
for i in range(len(fruits)):
    print('当前水果 :', fruits[i])
```

```
print("Good bye!")
```

结果：

```
当前水果 : banana
当前水果 : apple
当前水果 : mango
Good bye!
```

for 循环使用 else 语句

在 python 中，for … else ,for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

也就是说 for 循环中的 else 部分是循环结束后执行的，但如果循环被 break 结束后，else 部分便不再执行。

```
# coding=utf-8

for num in range(10, 20):  # 迭代 10 到 20 之间的数字
    for i in range(2, num):  # 根据因子迭代
        if num % i == 0:  # 确定第一个因子
            j = num / i  # 计算第二个因子
            print('%d 等于 %d * %d' % (num, i, j))
            break  # 跳出当前循环
    else:  # 循环的 else 部分
        print(num, '是一个质数')
```

结果：

```
10 等于 2 * 5
11 是一个质数
12 等于 2 * 6
13 是一个质数
14 等于 2 * 7
15 等于 3 * 5
16 等于 2 * 8
17 是一个质数
18 等于 2 * 9
19 是一个质数
```

计算出 100 以内的质数

例：

```
# coding=utf-8

prime = []
for num in range(2, 100):  # 迭代 2 到 100 之间的数字
    for i in range(2, num):  # 根据因子迭代
        if num % i == 0:  # 确定第一个因子
            break  # 跳出当前循环
    else:  # 循环的 else 部分
        prime.append(num)
print(prime)
```

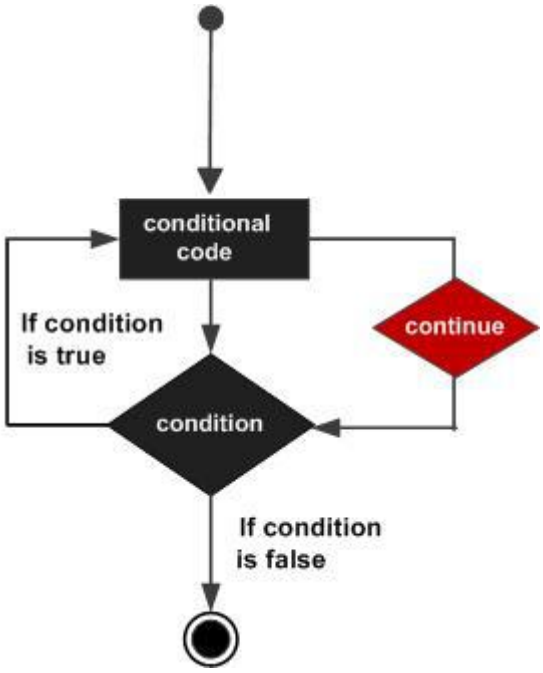
结果：

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

2.14.3. continue 语句

Python continue 语句跳出本次循环，而 break 跳出整个循环。
continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。
continue 语句用在 while 和 for 循环中。

流程图：



示例：

```
# -*- coding: utf-8 -*-

for letter in 'Python':  # 第一个实例
    if letter == 'h':
        continue

    print('当前字母 :', letter)


var = 5  # 第二个实例
while var > 0:
    var = var - 1
    if var == 2:
        continue

    print('当前变量值 :', var)

print("Good bye!")
```

结果：

```
当前字母 : P
```

```
当前字母 : y
当前字母 : t
当前字母 : o
当前字母 : n
当前变量值 : 4
当前变量值 : 3
当前变量值 : 1
当前变量值 : 0
Good bye!
```

2.14.4. python 打印矩形、菱形、三角形

实例代码：

```
# coding=utf-8

rows = int(input('输入列数： '))
# 实心正方形
print("实心正方形")
for y in range(rows):
    print(" * " * rows)
# 空心正方形
print("空心正方形")
for y in range(rows):
    for x in range(rows):
        if (x == 0 or x == rows - 1 or y == 0 or y == rows - 1):
            print(" * ", end="")
        else:
            print("   ", end="")
    print()

# 打印实心等边三角形
print("实心等腰直角三角形")
for y in range(rows):
    print(" * " * (rows - y))
# 打印空心等边三角形
print("空心等腰直角三角形")
for y in range(rows):
    for x in range(rows - y):
        if (x == 0 or x == rows - y - 1 or y == 0):
            print(" * ", end="")
        else:
            print("   ", end="")
    print()

# 打印实心等边三角形
print("打印实心等边三角形")
```

```
for n in range(1, rows + 1):  # 变量n控制行数
    print('   ' * (rows - n),end=" ")
    print(' * ' * (2 * n - 1))
# 打印空心等边三角形
print("打印空心等边三角形")
for n in range(1, rows + 1):  # 变量n控制行数
    print('   ' * (rows - n),end=" ")
    for x in range(2 * n - 1):
        if (x == 0 or x == 2 * n - 2 or n == rows):
            print(" * ", end="")
        else:
            print("   ", end="")
    print()

# 打印实心菱形
print("打印实心菱形")
for n in range(1, rows + 1):  # 变量n控制行数
    print('   ' * (rows - n),end="")
    print(' * ' * (2 * n - 1))
for n in range(1, rows):  # 变量n控制行数
    n = rows - n  # 反转n，用于打印倒置等边三角形
    print('   ' * (rows - n), end="")
    print(' * ' * (2 * n - 1))
# 打印空心菱形
print("打印空心菱形")
for n in range(1, rows + 1):  # 变量n控制行数
    print('   ' * (rows - n), end="")
    for x in range(2 * n - 1):
        if (x == 0 or x == 2 * n - 2):
            print(' * ', end="")
        else:
            print('   ', end="")
    print()
for n in range(1, rows):  # 变量n控制行数
    n = rows - n  # 反转n，用于打印倒置等边三角形
    print('   ' * (rows - n), end="")
    for x in range(2 * n - 1):
        if (x == 0 or x == 2 * n - 2):
            print(' * ', end="")
        else:
            print('   ', end="")
    print()
```

结果：

```
输入列数：  5
实心正方形
 *  *  *  *  *
 *  *  *  *  *
 *  *  *  *  *
 *  *  *  *  *
 *  *  *  *  *
```

空心正方形

```
* * * * *
*       *
*       *
*       *
* * * * *
```

实心等腰直角三角形

```
* * * * *
* * * *
* * *
* *
*
```

空心等腰直角三角形

```
* * * * *
*     *
*   *
* *
*
```

打印实心等边三角形

```
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
```

打印空心等边三角形

```
        *
      *   *
    *       *
  *           *
* * * * * * * * *
```

打印实心菱形

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * * *
    * * * * *
      * * *
      *
```

打印空心菱形

```
      *
    *   *
  *       *
*           *
  *       *
    *   *
      *
```

2.15. Python 数据类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。
以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

函数	描述
int(x [,base])	将 x 转换为一个整数
float(x)	将 x 转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效 Python 表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

2.15.1. 数值类型转换

int(x [,base]) 将 x 转换为一个整数
float(x) 将 x 转换到一个浮点数
complex(real [,imag]) 创建一个复数
chr(x) 将一个整数转换为一个字符
ord(x) 将一个字符转换为它的整数值
hex(x) 将一个整数转换为一个十六进制字符串
oct(x) 将一个整数转换为一个八进制字符串
bin(x) 将一个整数转换为一个二进制字符串
例：

```
print(int("1234"))
print(float(123),float("123"))
print(complex(10,2),complex(2,2))
print(hex(10))
```

```
print(oct(10))
print(bin(10))
```

结果：

```
1234
123.0 123.0
(10+2j) (2+2j)
0xa
0o12
0b1010
```

```
# chr(x)将一个整数转换为一个字符
# ord(x)将一个字符转换为它的整数值
print(ord('a'),chr(65))
```

结果：

```
97 A
```

2.15.2. tuple 和 list 的相互转换

例：

```
str = 'hello world'
print(list(str))
print(tuple(str))
tup = ('hello','world')
print(list(tup))
list = ['hello','world']
print(tuple(list))
```

结果：

```
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
['hello', 'world']
('hello', 'world')
```

2.15.3. 将 List 和 Tuple 复合数据类型转换为 Dictionary

```
li = ['name', 'age', 'city']
tup = ('jmilk', 23, 'BJ')
kvtup = zip(li, tup)
print(kvtup)
print(dict(kvtup))
```

结果：

```
[('name', 'jmilk'), ('age', 23), ('city', 'BJ')]
{'city': 'BJ', 'age': 23, 'name': 'jmilk'}
```

3. python 数据类型

在内存中存储的数据可以有多种类型。
例如，一个人的年龄可以用数字来存储，他的名字可以用字符串来存储。
Python 有五个标准的数据类型：
Numbers（数字）
String（字符串）
List（列表）
Tuple（元组）
Dictionary（字典）

3.1. 变量&常量

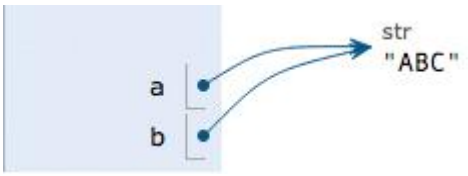
3.1.1. 变量在计算机内存中的表示

最后，理解变量在计算机内存中的表示也非常重要。当我们写：`a = 'ABC'`
时，Python 解释器干了两件事情：
1. 在内存中创建了一个`'ABC'`的字符串；
2. 在内存中创建了一个名为`a`的变量，并把它指向`'ABC'`。
也可以把一个变量`a`赋值给另一个变量`b`，这个操作实际上是把变量`b`指向变量`a`所指向的数据，例如下面的代码：

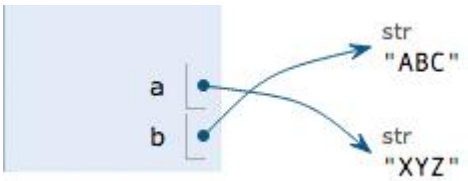
```
a = 'ABC'
b = a
a = 'XYZ'
print(b)
```

执行`a = 'ABC'`，解释器创建了字符串`'ABC'`和变量`a`，并把`a`指向`'ABC'`：

执行`b = a`，解释器创建了变量`b`，并把`b`指向`a`指向的字符串`'ABC'`：



执行`a = 'XYZ'`，解释器创建了字符串'XYZ'，并把`a`的指向改为`'XYZ'`，但`b`并没有更改：



所以，最后打印变量`b`的结果自然是`'ABC'`了。

3.1.2. 常量

所谓常量就是不能变的变量，比如常用的数学常数 π 就是一个常量。在 Python 中，通常用全部大写的变量名表示

常量：

```
PI = 3.14159265359
```

但事实上 PI 仍然是一个变量，Python 根本没有任何机制保证 PI 不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量 PI 的值，也没人能拦住你。

3.2. Number 数字

数字数据类型用于存储数值。
他们是不可改变的数据类型，这意味着改变数字数据类型会分配一个新的对象。
当你指定一个值时，Number 对象就会被创建：

```
var1 = 1
var2 = 10
```

您可以通过使用 del 语句删除单个或多个对象的引用。例如：

```
del var1
del var1,var2
```

数字以 0o 开头表示 8 进制所有数字最大为 7，以 0x 口头表示 16 进制数字，例：

```
print(10, 0o70, 0x3C)
```

运行结果：
10 56 60

Python 支持支持三种不同的数值类型：

- 整型(Int) - 通常被称为是整型或整数，是正或负整数，不带小数点。Python3 整型是没有限制大小的，可以当作 Long 类型使用，所以 Python3 没有 Python2 的 Long 类型。
- 浮点型(float) - 浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示（2.5e2 = 2.5 x 10^2 = 250）
- 复数((complex)) - 复数由实数部分和虚数部分构成，可以用 a + bj,或者 complex(a,b)表示， 复数的实部 a 和虚部 b 都是浮点型。

一些数值类型的实例：

int	float	complex
10	0	3.14j
100	15.2	45.j
-786	-21.9	9.322e-36j
0o70	3.23E+19	.876j
-470	-90	-.6545+0J
-0x260	-3.25E+101	3e+26J
0x69	7.02E-11	4.53e-7j

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

浮点数只支持十进制.

例：

```
intvar1,intvar2,intvar3=10,-0x260,0o470
floatvar1,floatvar2,floatvar3=15.20,32.3e+18,-90.
complexvar1,complexvar2,complexvar3=3.14j,3e+26J,complex(3,26)

print(intvar1,intvar3,intvar3)
print(floatvar1,floatvar2,floatvar3)
print(complexvar1,complexvar2,complexvar3)
```

运行结果：

```
10 312 312
15.2 3.23e+19 -90.0
3.14j 3e+26j (3+26j)
```

3.2.1. Python 数学函数

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如 abs(-10) 返回 10
fabs(x)	返回数字浮点形式的绝对值，如 math.fabs(-10) 返回 10.0
floor(x)	返回数字的下舍整数，如 math.floor(4.9)返回 4
ceil(x)	返回数字的上入整数，如 math.ceil(4.1) 返回 5
cmp(x, y) **(x>y)-(x<y)**	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1**Python 3 已废弃，使用 (x>y)-(x<y) 替换**。
exp(x)	返回 e 的 x 次幂(e^x),如 math.exp(1) 返回 2.718281828459045
log(x)	如 math.log(math.e)返回 1.0,math.log(100,10)返回 2.0
log10(x)	返回以 10 为基数的 x 的对数，如 math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回 x 的整数部分与小数部分，两部分的数值符号与 x 相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数 x 的四舍五入值，如给出 n 值，则代表舍入到小数点后的位数。
sqrt(x)	返回数字 x 的平方根，数字可以为负数，返回类型为实数，如 math.sqrt(4)返回 2+0j

abs 与 fabs 函数实例

```
# coding=utf-8

print("abs() 函数返回数字的绝对值。")
print("abs(-45) : ", abs(-45))
print("abs(100.12) : ", abs(100.12))

import math   # 导入数学模块
print("math.fabs(-45.17) : ", math.fabs(-45.17))
print("math.fabs(100.12) : ", math.fabs(100.12))
print("math.fabs(100.72) : ", math.fabs(100.72))
print("math.fabs(math.pi) : ", math.fabs(math.pi))
```

结果：

```
abs() 函数返回数字的绝对值。
abs(-45) :  45
abs(100.12) :  100.12
math.fabs(-45.17) :  45.17
math.fabs(100.12) :  100.12
math.fabs(100.72) :  100.72
math.fabs(math.pi) :  3.141592653589793
```

floor() 函数实例

```
import math   # This will import math module

print("math.floor(-45.17) : ", math.floor(-45.17))
print("math.floor(100.12) : ", math.floor(100.12))
print("math.floor(100.72) : ", math.floor(100.72))
print("math.floor(math.pi) : ", math.floor(math.pi))
```

结果：

```
math.floor(-45.17) :  -46
math.floor(100.12) :  100
math.floor(100.72) :  100
math.floor(math.pi) :  3
```

ceil() 函数实例

```
import math

print("ceil() 函数返回数字的上入整数")
print("math.ceil(-45.17) : ", math.ceil(-45.17))
print("math.ceil(100.12) : ", math.ceil(100.12))
print("math.ceil(100.72) : ", math.ceil(100.72))
print("math.ceil(math.pi) : ", math.ceil(math.pi))
```

结果：

```
ceil() 函数返回数字的上入整数
math.ceil(-45.17) :  -45
```

```
math.ceil(100.12) :  101
math.ceil(100.72) :  101
math.ceil(math.pi) :  4
```

exp() 方法实例

```
import math

print("math.exp(-45.17) : ", math.exp(-45.17))
print("math.exp(100.12) : ", math.exp(100.12))
print("math.exp(100.72) : ", math.exp(100.72))
print("math.exp(math.pi) : ", math.exp(math.pi))
```

结果：

```
math.exp(-45.17) :  2.4150062132629406e-20
math.exp(100.12) :  3.0308436140742566e+43
math.exp(100.72) :  5.522557130248187e+43
math.exp(math.pi) :  23.140692632779267
```

log 函数实例

```
import math

print("math.log(100.12) : ", math.log(100.12))
print("math.log(100.72) : ", math.log(100.72))
print("math.log(math.pi) : ", math.log(math.pi))
print("math.log(math.pi) : ", math.log(math.pi, math.e))

print("math.log10(100.12) : ", math.log10(100.12))
print("math.log10(100.72) : ", math.log10(100.72))
print("math.log10(math.pi) : ", math.log10(math.pi))
```

结果：

```
math.log(100.12) :  4.6063694665635735
math.log(100.72) :  4.612344389736092
math.log(math.pi) :  1.1447298858494002
math.log(math.pi) :  1.1447298858494002

math.log10(100.12) :  2.0005208409361854
math.log10(100.72) :  2.003115717099806
math.log10(math.pi) :  0.49714987269413385
```

其他函数实例

```
import math

print("math.modf(100.72) : ", math.modf(100.72))
print("math.modf(math.pi) : ", math.modf(math.pi))
print("round(80.23456, 2) : ", round(80.23456, 2))
print("round(math.pi, 3) : ", round(math.pi, 3))
```

结果：

```
math.modf(100.72) :  (0.7199999999999989, 100.0)
math.modf(math.pi) :  (0.14159265358979312, 3.0)
round(80.23456, 2) :  80.23
round(math.pi, 3) :  3.142
```

3.2.2. Python 随机数函数

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

实例：

```
# coding=utf-8

import random

#返回一个列表的随机项。
print("choice([1, 2, 3, 5, 9]) : ", random.choice([1, 2, 3, 5, 9]))
#返回一个字符串的随机字符。
print("choice('A String') : ", random.choice('A String'))
# 输出 100 <= number < 1000 间的偶数
print("randrange(100, 1000, 2) : ", random.randrange(100, 1000, 2))
# 输出 100 <= number < 1000 间的其他数
print("randrange(100, 1000, 3) : ", random.randrange(100, 1000, 3))
# 生成一个[0,1)范围的实数
print("random():", random.random())

# seed() 方法改变随机数生成器的种子，可以在调用其他随机模块函数之前调用此函数。
```

```
random.seed( 10 )
print("Random number with seed 10 : ", random.random())
random.seed( 10 )
print("Random number with seed 10 : ", random.random())

#将序列的所有元素随机排序
list = [20, 16, 10, 5]
random.shuffle(list)
print("随机排序列表 : ",  list)

#生成一个指定范围的随机数
print("uniform(5, 10) 的随机数为 : ",  random.uniform(5, 10))
```

结果：

```
choice([1, 2, 3, 5, 9]) :  5
choice('A String') :  t
randrange(100, 1000, 2) :  982
randrange(100, 1000, 3) :  709
random(): 0.9091187307230327
Random number with seed 10 :  0.5714025946899135
Random number with seed 10 :  0.5714025946899135
随机排序列表 :  [10, 20, 16, 5]
uniform(5, 10) 的随机数为 :  6.030491160697508
```

3.2.3. Python 三角函数

函数	描述
acos(x)	返回 x 的反余弦弧度值。 x -- -1 到 1 之间的数值。如果 x 是大于 1，会产生一个错误
asin(x)	返回 x 的反正弦弧度值。 x -- -1 到 1 之间的数值。如果 x 是大于 1，会产生一个错误。
atan(x)	返回 x 的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回 x 的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(x*x + y*y)。
sin(x)	返回的 x 弧度的正弦值。
tan(x)	返回 x 弧度的正切值。
degrees(x)	将弧度转换为角度,如 degrees(math.pi/2) ， 返回 90.0

radians(x)	将角度转换为弧度

```
import math

print("acos(0.64) : ",  math.acos(0.64))
print("asin(0.64) : ",  math.asin(0.64))
print("atan(10) : ",  math.atan(10))
print("atan2(5,5) : ",  math.atan2(5,5))
print("cos(3) : ",  math.cos(3))
print("hypot(3, 4) : ",  math.hypot(3, 4))
print("sin(math.pi/2) : ",  math.sin(math.pi/2))
print("tan(math.pi/4) : ",  math.tan(math.pi/4))
print("degrees(math.pi/2) : ",  math.degrees(math.pi/2))
print("radians(math.pi) : ",  math.radians(math.pi))
```

结果：

```
acos(0.64) :  0.8762980611683406
asin(0.64) :  0.694498265626556
atan(10) :  1.4711276743037347
atan2(5,5) :  0.7853981633974483
cos(3) :  -0.9899924966004454
hypot(3, 4) :  5.0
sin(math.pi/2) :  1.0
tan(math.pi/4) :  0.9999999999999999
degrees(math.pi/2) :  90.0
radians(math.pi) :  0.05483113556160755
```

Python 数学常量

```
import math

print(math.pi, math.e)
```

结果：

3.14159265359 2.71828182846

3.3. str 字符串

一般字符串表示为

str="a1a2・・・an"(n>=0)

str 表示一个长度为 n 的字符串

python 字符串的下标是从 0 开始的，一直到 n-1 结束

"a1a2・・・an"每个字符对应的角标为"0,1,...,n-1"

使用 str[头下标:尾下标]，就可以截取相应的字符串（不包括尾角标的字符，即含头不含尾）

例：

```
str = 'Hello World!'
print(str)  # 输出完整字符串
```

```
print(str[0])  # 输出字符串中的第一个字符
print(str[2:5])  # 输出字符串中第三个至第五个之间的字符串
print(str[2:])  # 输出从第三个字符开始的字符串,等价于print(str[2:len(str)])
print(str * 2)  # 输出字符串两次
print(str + "TEST")  # 输出连接的字符串
```

运行结果：

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

3.3.1. 字符串转义字符

在需要在字符中使用特殊字符时，python 用反斜杠(\)转义字符。如下表：

转义字符	描述
\(在行尾时)	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy 代表的字符，例如：\o12 代表换行
\xyy	十六进制数，yy 代表的字符，例如：\x0a 代表换行

转义字符	描述
\other	其它的字符以普通格式输出

3.3.2. 字符串运算符

a="Hello"，b="Python"：

操作符	描述	实例
+	字符串连接	>>>a + b 'HelloPython'
*	重复输出字符串	>>>a * 2 'HelloHello'
[]	通过索引获取字符串中字符	>>>a[1] 'e'
[:]	截取字符串中的一部分	>>>a[1:4] 'ell'
in	成员运算符 - 如果字符串中包含给定的字符返回 True	>>>"H" in a TRUE
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	>>>"M" not in a TRUE
r/R	原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。	print(r'\n')或 print(R'\n') \n
%s	格式化字符串的输出。在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法	>>>print("My name is %s and weight is %d kg!" % ('Zhangsan', 21)) My name is Zhangsan and weight is 21 kg!

3.3.3. 字符串格式化

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

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

python 字符串格式化符号:

符 号	描述
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数

%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f 和%e 的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令:

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零('0'),在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

3.3.4. str.format()

```
# coding=utf-8

print("{} {}".format("hello", "world"))  # 不设置指定位置，按默认顺序
print("{0} {1}".format("hello", "world"))  # 设置指定位置
print("{1} {0} {1}".format("hello", "world"))  # 设置指定位置

# 设置参数

print("网站名：{name}, 地址 {url}".format(name="百度", url="www.baidu.com"))

# 通过字典设置参数
```

```
site = {"name": "百度", "url": "www.baidu.com"}

print("网站名：{name}, 地址 {url}".format(**site))

# 通过列表索引设置参数

my_list = ['百度', 'www.baidu.com']

print("网站名：{0[0]}, 地址 {0[1]}".format(my_list))  # "0" 是可选的
```

结果：

```
hello world
hello world
world hello world
网站名：百度, 地址 www.baidu.com
网站名：百度, 地址 www.baidu.com
网站名：百度, 地址 www.baidu.com
```

3.3.5. str.format() 格式化数字

下表展示了 str.format() 格式化数字的多种方法：

数字	格式	输出	描述
3.1415926	{:.2f}	3.14	保留小数点后两位
3.1415926	{:+.2f}	3.14	带符号保留小数点后两位
-1	{:+.2f}	-1	带符号保留小数点后两位
2.71828	{:.0f}	3	不带小数
5	{:0>2d}	5	数字补零 (填充左边, 宽度为 2)
5	{:x<4d}	5xxx	数字补 x (填充右边, 宽度为 4)
10	{:x<4d}	10xx	数字补 x (填充右边, 宽度为 4)
1000000	{:,}	1,000,000	以逗号分隔的数字格式
0.25	{:.2%}	25.00%	百分比格式
1000000000	{:.2e}	1.00E+09	指数记法
13	{:>10d}	13	右对齐 (默认, 宽度为 10)
13	{:<10d}	13	左对齐 (宽度为 10)
13	{:^10d}	13	中间对齐 (宽度为 10)

```
>>> print("{:.2f}".format(3.1415926));

3.14
```

`^`, `<`, `>` 分别是居中、左对齐、右对齐，后面带宽度， `:` 号后面带填充的字符，只能是一个字符，不指定则默认是用空格填充。

`+` 表示在正数前显示 `+`，负数前显示 `-`； （空格）表示在正数前加空格

b、d、o、x 分别是二进制、十进制、八进制、十六进制。

此外我们可以使用大括号 `{}` 来转义大括号，如下实例：

```
print("{} 对应的位置是 {{0}}".format("runoob"))
```

输出结果为：

```
runoob 对应的位置是 {0}
```

3.3.6. Unicode 字符串

在 Python2 中，引号前小写的"u"表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 Unicode-Escape 编码。如下例所示：

```
>>>print(u'Hello\u0020World !')
Hello World !
```

被替换的 \u0020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符（空格符）。

在 Python3 中，所有的字符串都是 Unicode 字符串，不需要加 u 标示。

```
>>>print('Hello\u0020World !')
Hello World !
```

3.3.7. 字符串内建函数

所有的方法都包含了对 Unicode 的支持，有一些甚至是专门用于 Unicode 的。

方法	描述
string.capitalize()	将字符串的第一个字母变成大写,其他字母变小写
string.center(width[, fillchar])	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
bytes.decode(encoding="utf-8", errors="strict")	Python3 中没有 decode 方法，但我们可以使用 bytes 对象的 decode() 方法来解码给定的 bytes 对象，这个 bytes 对象可以由 str.encode() 来编码返回。
string.encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0,	检查字符串是否以 obj 结束，如果 beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.

end=len(string))	
string.expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8。
string.find(str, beg=0, end=len(string))	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.format()	格式化字符串
string.index(str, beg=0, end=len(string))	跟 find()方法一样，只不过如果 str 不在 string 中会报一个异常.
string.isalnum()	如果 string **所有字符都是字母或数字**并且至少有一个字符，则返回 True,否则返回 False
string.isalpha()	如果 string **所有字符都是字母**并且至少有一个字符则返回 True,否则返回 False
string.isdigit()	如果 string **只包含数字**则返回 True 否则返回 False.
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符**都是小写**，则返回 True，否则返回 False
string.isspace()	如果 string 中只包含空格，则返回 True，否则返回 False.
string.istitle()	如果所有的单词拼写首字母是否为大写，且其他字母为小写则返回 True，否则返回 False
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq)	以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width[, fillchar])	返回一个原字符串左对齐,并使用空格或指定字符填充至长度 width 的新字符串
string.lower()	转换 string 中所有大写字符为小写.
string.lstrip([chars])	截掉字符串左边的空格或指定字符

max(str)	返回字符串 str 中最大的字母。
min(str)	返回字符串 str 中最小的字母。
string.partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把字符串 string 分成一个 3 元素的元组(string_pre_str,str,string_post_str),如果 string 中不包含 str 则 string_pre_str == string.
string.rpartition(str)	类似于 partition()函数,不过是从右边开始查找.
string.replace(str1, str2, num=string.count(str1))	把 string 中的 str1 替换成 str2,如果指定 num ，则替换不超过 num 次.
string.rfind(str, beg=0,end=len(string))	返回字符串最后一次出现的位置(从右向左查询)，如果没有匹配项则返回-1
string.rindex(str, beg=0,end=len(string))	返回子字符串 str 在字符串中最后出现的位置，如果没有匹配的字符串会报异常，可以指定可选参数[beg:end]设置查找的区间
string.rjust(width[, fillchar])	返回一个原字符串右对齐,并使用空格或指定的字符填充至长度 width 的新字符串
string.rstrip([chars])	删除 string 字符串末尾的空格或指定字符.
string.split(str="",num=string.count(str))	以 str 为分隔符切片 string，如果 num 有指定值，则仅分隔 num 个子字符串
string.splitlines([keepends])	按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
string.startswith(obj, beg=0,end=len(string))	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果 beg 和 end 指定值，则在指定范围内检查.
string.strip([chars])	用于移除字符串头尾指定的字符（默认为空格），相等于在 string 上执行 lstrip()和 rstrip()

string.swapcase()	反转 string 中的大小写
string.title()	转换字符串为所有单词都是以大写开始，其余字母均为小写
string.upper()	转换 string 中的小写字母为大写
string.zfill(width)	返回长度为 width 的字符串，原字符串 string 右对齐，前面填充 0
string.isdecimal()	isdecimal()方法检查字符串是否只包含十进制字符。 这个方法只存在于 unicode 字符串。

3.3.8. Maketrans&translate

注：Python3.4 已经没有 string.maketrans() 了，取而代之的是内建函数: bytearray.maketrans()、bytes.maketrans()、str.maketrans() 。

maketrans()方法用于创建字符映射的转换表：

语法：str.maketrans(intab, outtab)

参数：

intab -- 字符串中要替代的字符组成的字符串。

outtab -- 相应的映射字符的字符串。

两个字符串的长度必须相同，为一一对应的关系。

translate() 方法根据参数 table 给出的表(包含 256 个字符)转换字符串的字符,要过滤掉的字符放到 deletechars 参数中。

translate()方法语法：

str.translate(table)

bytes.translate(table[, delete])

bytearray.translate(table[, delete])

参数：

table -- 翻译表，翻译表是通过 maketrans() 方法转换而来。

deletechars -- 字符串中要过滤的字符列表。

返回翻译后的字符串,若给出了 delete 参数，则将原来的 bytes 中的属于 delete 的字符删除，剩下的字符要按照 table 中给出的映射来进行映射 。

例：

```
intab = "aeiou"
outtab = "12345"
trantab = str.maketrans(intab, outtab)
print(trantab)

str = "this is string example....wow!!!"
#根据 trantab 转换表转换字符串的字符
print(str.translate(trantab))
```

结果：

```
{97: 49, 101: 50, 105: 51, 111: 52, 117: 53}
th3s 3s str3ng 2x1mpl2....w4w!!!
```

```
# 转换为大写，并删除字母o
bytes_tabtrans = bytes.maketrans(b'abcdefghijklmnopqrstuvwxyz', b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
print(b'lenovo'.translate(bytes_tabtrans, b'o'))
```

结果：

```
b'LENV'
```

3.3.9. 字符串格式化示例

输出一个平方与立方的表:

```
for x in range(1, 11):
    print(str(x).rjust(2), str(x * x).rjust(3), end=' ')
    print(str(x * x * x).rjust(4))

for x in range(1, 11):
    print('{0:2d} {1:3d} {2:4d}'.format(x, x * x, x * x * x))

for x in range(1, 11):
    print('%2d %3d %4d' % (x, x * x, x * x * x))
```

结果：

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

美化表格：

```
table = {'Google': 1, 'Baidu': 2, 'Taobao': 3, 'Tencent': 4}
for name, number in table.items():
    print('{:10} ==> {:10d}'.format(name, number))
```

结果：

```
Google     ==>          1
Baidu      ==>          2
Taobao     ==>          3
Tencent    ==>          4
```

3.4. list 列表

List（列表） 是 Python 中使用最频繁的数据类型。
列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。
列表用[ ]标识。是 python 最通用的复合数据类型。
列表中的值得分割也可以用到变量[头下标:尾下标]，就可以截取相应的列表，从左到右索引从 0 开始的（跟字符串一样）。
加号（+）是列表连接运算符，星号（*）是重复操作。如下实例：

```
#coding=utf-8

tinylist = [123, 'john']
list = [ 'hello', 786 , 2.23, 'world', 70.2,tinylist]

list[1]='rewrites'      # list 的值可以修改
print(list)              # 输出完整列表
print(list[0])           # 输出列表的第一个元素
print(list[1])           # 输出列表的第二个元素
print(list[1:3])         # 输出第二个至第三个的元素
print(list[2:])          # 输出从第三个开始至列表末尾的所有元素
print(tinylist * 2)      # 输出列表两次
print(list + tinylist)   # 打印组合的列表
```

运行结果：

```
['hello', 'rewrites', 2.23, 'world', 70.2, [123, 'john']]
hello
rewrites
['rewrites', 2.23]
[2.23, 'world', 70.2, [123, 'john']]
[123, 'john', 123, 'john']
['hello', 'rewrites', 2.23, 'world', 70.2, [123, 'john'], 123, 'john']
```

3.4.1. 删除列表元素

可以使用 del 语句来删除列表的的元素，如下实例：

```
list1 = ['physics', 'chemistry', 1997, 2000]

print(list1)
del list1[2]

print("After deleting value at index 2 : ")
print(list1)
```

结果：

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

3.4.2. list 的操作符

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

Python 表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print(x,end=" ")	1 2 3	迭代

3.4.3. list 列表截取

L[m:n]表示：从索引 m 开始直到索引 n(不包含 n)取数据;
L[m:n:p]表示：从索引 m 开始直到索引 n(不包含 n)取数据，每 p 个取一个。

1、m、n、p、可以为负数：
1、p>0 从首部往尾部方向取，p<0 是从尾部向首部取。
2、m、n 大于 0 表示索引值。
3、m、n 小于 0 表示倒数第几个。
L[-5:-2] 表示倒数第 5 到倒数第 2 元素（不包含倒数第 2 个元素）从首到尾的顺序取。
L[-2:-5:-2] 表示倒数第 2 到倒数第 5 元素（不包含倒数第 5 个元素）从尾到首的顺序取，每 2 个取一个。
一般流程：m、n 如果为负数先转为正数然后根据 p 的方向取值。

2、m、n、p 可以缺省：L[::]
m 缺省为 p 表示的方向的首元素，n 缺省为 p 表示的方向的尾元素的后一个，p 缺省表示 1

L = ['Google', 'Baidu', 'Taobao', 'edu360', 'xiaoniu']

Python 表达式	结果	描述
L[2]	'Taobao'	读取列表中第三个元素
L[-2]	'edu360'	读取列表中倒数第二个元素
L[1:]	['Baidu', 'Taobao', 'edu360', 'xiaoniu']	从第二个元素开始截取列表
L[::2]	['Google', 'Taobao', 'xiaoniu']	从开头到结尾，每隔 2 个元素读一个
L[-4:-1]	['Baidu', 'Taobao', 'edu360']	读取列表中倒数第四个元素到倒数第二个元素

L[-2:-5:-2]	['edu360', 'Baidu']	读取倒数第 2 到倒数第 4 个元素，每隔两个元素读一个
L[-4:-1:2]	['Baidu', 'edu360']	读取倒数第 4 到倒数第 2 个元素，每隔两个元素读一个

3.4.4. list 内部方法

方法	描述
list.append(obj)	在列表末尾添加新的对象
list.count(obj)	统计某个元素在列表中出现的次数
list.extend(seq)	在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
list.index(obj)	从列表中找出某个值第一个匹配项的索引位置
list.insert(index, obj)	将对象插入列表
list.pop(obj=list[-1])	移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
list.remove(obj)	移除列表中某个值的第一个匹配项
list.reverse()	反向列表中元素
list.sort([func])	对原列表进行排序

实例：

```
aList = [123, 'xyz', 'zara', 'abc']
aList.append(123)
print("Updated List : ", aList)

print("Count for 123 : ", aList.count(123))
print("Count for zara : ", aList.count('zara'))

aList.extend([2009, 'manni'])
print("Extended List : ", aList)

print("Index for xyz : ", aList.index('xyz'))
print("Index for zara : ", aList.index('zara'))

aList.insert(3, 2009)
print("Final List : ", aList)

print("A List : ", aList.pop())
print("B List : ", aList.pop(2))

print(aList)
aList.remove('xyz')
print("List : ", aList)
aList.remove('abc')
print("List : ", aList)
```

```
aList.reverse()
print("List : ", aList)

aList.sort()
print("List : ", aList)
```

结果:

```
Updated List :  [123, 'xyz', 'zara', 'abc', 123]
Count for 123 :  2
Count for zara :  1
Extended List :  [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
Index for xyz :  1
Index for zara :  2
Final List :  [123, 'xyz', 'zara', 2009, 'abc', 123, 2009, 'manni']
A List :  manni
B List :  zara
[123, 'xyz', 2009, 'abc', 123, 2009]
List :  [123, 2009, 'abc', 123, 2009]
List :  [123, 2009, 123, 2009]
List :  [2009, 123, 2009, 123]
List :  [123, 123, 2009, 2009]
```

3.4.5. 生成 list 相关函数

3.4.5.1. range 函数 list 序列迭代对象

Python3 range() 函数返回的是一个可迭代对象（类型是对象），而不是列表类型， 所以打印的时候不会打印列表。
Python3 list() 函数是对象迭代器，可以把 range()返回的可迭代对象转为一个列表，返回的变量类型为列表。
Python2 range() 函数返回的是列表。
range 语法：

range(stop)
range(start, stop[, step])

参数说明：

- start: 计数从 start 开始。默认是从 0 开始。例如 range（5）等价于 range（0， 5）;
- end: 计数到 end 结束，但不包括 end。例如：range（0， 5） 是[0, 1, 2, 3, 4]没有 5
- step：步长，默认为 1。例如：range（0， 5） 等价于 range(0, 5, 1)

```
# coding=utf-8

print(list(range(10)))  # 产生一个 0-9 的序列
print(list(range(3, 10)))  # 产生一个 3-9 的序列
print(list(range(0, 10, 3)))  # 产生从 0 开始，按 3 递增，最大值为 9 的序列
print(list(range(10, -1, -1)))  # 产生从 10 开始,最小值为 0 的递减序列
```

结果：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7, 8, 9]
[0, 3, 6, 9]
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

3.4.5.2. 列表生成式

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

```
#coding=utf-8

print([x * x for x in range(1, 11)])
print([x * x for x in range(1, 11) if x % 2 == 0])

#还可以使用两层循环，可以生成全排列：
print([m + n for m in 'ABC' for n in 'XYZ'])
print([str(x)+str(y) for x in range(1,6) for y in range(11,16)])

#for 循环其实可以同时使用两个甚至多个变量，比如 dict 的 items()可以同时迭代 key 和 value：
d = {'x': 'A', 'y': 'B', 'z': 'C' }
print([k + '=' + v for k, v in d.items()])
```

结果：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[4, 16, 36, 64, 100]
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
['111', '112', '113', '114', '115', '211', '212', '213', '214', '215', '311', '312', '313', '314', '315', '411', '412', '413', '414', '415',
'511', '512', '513', '514', '515']
['x=A', 'y=B', 'z=C']
```

3.4.5.3. 列表生成器

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

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

只要把一个列表生成式的[]改成()，就创建了一个 generator：

g = (x * x for x in range(10))

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

实例：

```
# coding=utf-8

for x in (x * x for x in range(1, 11)):
    print(x, end=" ")
print()
for x in (x * x for x in range(1, 11) if x % 2 == 0):
    print(x, end=" ")
```

结果：

```
1 4 9 16 25 36 49 64 81 100
4 16 36 64 100
```

3.4.5.4. 函数列表生成器

如果推算的算法比较复杂，用类似列表生成式的 `for` 循环无法实现的时候，还可以用函数来实现。
比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：
1，1，2，3，5，8，13，21，34，...
斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
```

上面的函数可以输出斐波那契数列的前 N 个数：

```
>>> fib(6)
1
1
2
3
5
8
```

仔细观察，可以看出，`fib` 函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似 generator。
也就是说，上面的函数和 generator 仅一步之遥。要把 `fib` 函数变成 generator，只需要把 `print(b)` 改为 `yield b` 就可以了：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
```

这就是定义 generator 的另一种方法。如果一个函数定义中包含 `yield` 关键字，那么这个函数就不再是一个普通函数，而是一个 generator：

```
>>> fib(6)
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是 generator 和函数的执行流程不一样。函数是顺序执行，遇到 return 语句或者最后一行函数语句就返回。而变成 generator 的函数，在每次调用 `next()` 的时候执行，遇到 `yield` 语句返回，再次执行时从上次返回的 `yield` 语句处继续执行。
举个简单的例子，定义一个 generator，依次返回数字 1，3，5：

```
def odd():
    print('step 1')
    yield 1
    print('step 2')
    yield(3)
    print('step 3')
    yield(5)
```

调用该 generator 时，首先要生成一个 generator 对象，然后用 next()函数不断获得下一个返回值：

```
>>> o = odd()
```

```
>>> next(o)

step 11

>>> next(o)

step 23

>>> next(o)

step 35

>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到，`odd`不是普通函数，而是 generator，在执行过程中，遇到`yield`就中断，下次又继续执行。执行 3 次`yield`后，已经没有`yield`可以执行了，所以，第 4 次调用`next()`就抛出`StopIteration`异常。

可以 for 循环迭代 generator：

```
for i in odd():
    print(i)
```

结果：

```
step 1
1
step 2
3
step 3
5
```

回到`fib`的例子，我们在循环过程中不断调用`yield`，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成 generator 后，我们基本上从来不会用`next()`来调用它，而是直接使用`for`循环来迭代：

```
for n in fib(6):
    print(n)
```

generator 是在`for`循环的过程中不断计算出下一个元素，并在适当的条件结束`for`循环。对于函数改成的 generator 来说，遇到 return 语句或者执行到函数体最后一行语句，就是结束 generator 的指令，`for`循环随之结束。

3.5. tuple 元组

元组是另一个数据类型，类似于 List（列表）。[]

元组用"()"标识。内部元素用逗号隔开。但是元组中的元素值是**不允许修改，相当于只读列表**。

例：

```
# coding=utf-8

tinytuple = (123, 'john')
tuple = ('hello', 786, 2.23, 'world', 70.2, tinytuple)
```

```
# tuple[1]= 'rewrites'      tuple 的值不可以修改，否则运行时报错
print(tuple)  # 输出完整元组
print(tuple[0])  # 输出元组的第一个元素
print(tuple[1])  # 输出元组的第二个元素
print(tuple[1:3])  # 输出第二个至第三个的元素
print(tuple[2:])  # 输出从第三个开始至元组末尾的所有元素
print(tinytuple * 2)  # 输出元组两次
print(tuple + tinytuple)  # 打印组合的元组
```

结果：

```
('hello', 786, 2.23, 'world', 70.2, (123, 'john'))
hello
786
(786, 2.23)
(2.23, 'world', 70.2, (123, 'john'))
(123, 'john', 123, 'john')
('hello', 786, 2.23, 'world', 70.2, (123, 'john'), 123, 'john')
```

3.5.1. 元组申明

Python 的元组与列表类似，不同之处在于元组的元素不能修改。
元组使用小括号，列表使用方括号。
元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。
如下实例：

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

创建空元组

```
tup1 = ()
```

元组中只包含一个元素时，需要在元素后面添加逗号，否则会忽略括号，不识别为元组

```
tup1 = (50,)
```

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
tuple = 'abc', -4.24e93, 18 + 6.6j, 'xyz'
print(tuple, type(tuple))
x, y = 1, 2
print("Value of x , y : ", x, y)
```

结果：

```
('abc', -4.24e+93, (18+6.6j), 'xyz') <class 'tuple'>
Value of x , y :  1 2
```

3.5.2. 删除元组

元组中的元素值是不允许删除的，但我们可以使用 del 语句来删除整个元组，如下实例:

```
tup = ('physics', 'chemistry', 1997, 2000)

print(tup)
del tup
print("After deleting tup : ")
print(tup)
```

3.5.3. 元组运算符

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

Python 表达式	结果	描述
len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
3 in (1, 2, 3)	True	元素是否存在
for x in (1, 2, 3): print(x, end=" ")	1 2 3	迭代

3.5.4. 元组索引，截取

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素，如下所示：

L = ('spam', 'Spam', 'SPAM!')

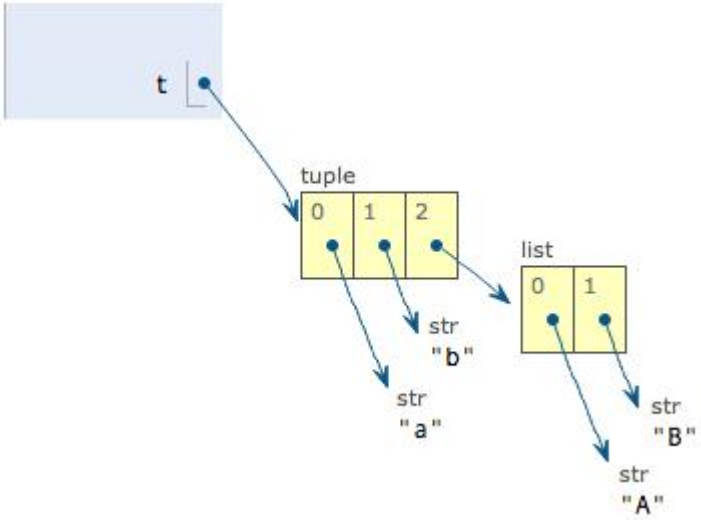
表达式	结果	描述
L[2]	'SPAM!'	读取第三个元素
L[-2]	'Spam'	反向读取；读取倒数第二个元素
L[1:]	('Spam', 'SPAM!')	截取元素

3.5.5. “可变的”tuple

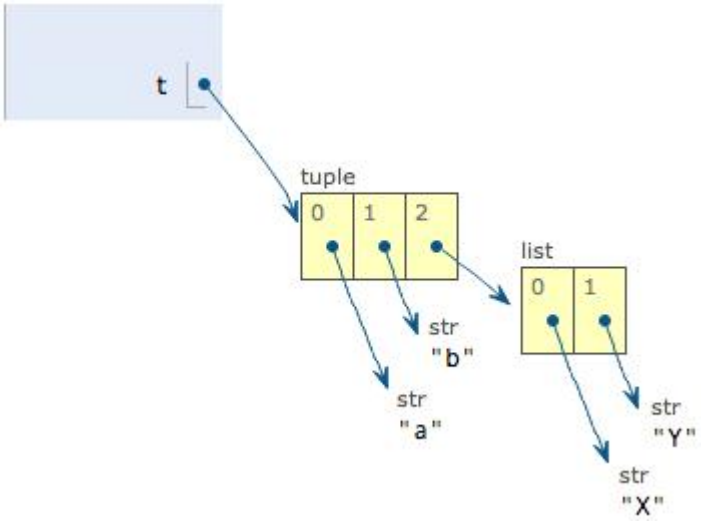
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

tuple 的每个元素指向是不能改变的，但指向的可变元素是可变的！

这个 tuple 定义的时候有 3 个元素，分别是'a'，'b'和一个 list:



当我们把 list 的元素'A'和'B'修改为'X'和'Y'后，tuple 变为：



3.6. dict 字典

字典(dictionary)是**键值对形式的可变容器**，可存储任意类型对象。列表是有序的对象结合，字典是无序的对象集合。字典的每个键值(key=>value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号({})中

字典键的特性

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，前一个的值会被后一个覆盖

2）值可以取任何数据类型，但键必须是不可变的，如字符串，数字或元组。因为列表是可变的，所以键不能为列表

例：

```
#coding=utf-8

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print(dict)                # 输出完整的字典
print(tinydict)            # 输出完整的字典
```

```
print(dict['one'])          # 输出键为'one' 的值
print(dict[2])              # 输出键为 2 的值
print(tinydict.keys())      # 以 list 形式, 输出所有键
print(tinydict.values())    # 以 list 形式, 输出所有值
print(tinydict.items())     # 以 list 形式, 输出所有键值对元组
```

结果：

```
{'one': 'This is one', 2: 'This is two'}
{'name': 'john', 'code': 6734, 'dept': 'sales'}
This is one
This is two
dict_keys(['name', 'code', 'dept'])
dict_values(['john', 6734, 'sales'])
dict_items([('name', 'john'), ('code', 6734), ('dept', 'sales')])
```

3.6.1. 修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8  # update existing entry
dict['School'] = "DPS School"  # Add new entry

print("dict['Age']: ", dict['Age'])
print("dict['School']: ", dict['School'])
```

结果：

```
dict['Age']:  8
dict['School']:  DPS School
```

3.6.2. 删除字典元素

能删单一的元素也能清空字典，清空只需一项操作。
显示删除字典的一个键用 del 命令，如下实例：

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']  # 删除键是'Name'的条目
dict.clear()      # 清空词典所有条目
del dict          # 删除词典
```

3.6.3. 字典内置方法

函数	描述
dict.clear()	删除字典内所有元素
dict.copy()	返回一个字典的浅复制
dict.fromkeys(seq[, val]))	创建一个新字典，以序列 seq 中元素做字典的键，val 为字典所有键对应的初始值
dict.get(key,default=None)	返回指定键的值，如果值不在字典中返回 default 值
key in dict	如果键在字典 dict 里返回 true，否则返回 false
dict.items()	以列表返回可遍历的(键, 值) 元组数组
dict.keys()	以列表返回一个字典所有的键
dict.setdefault(key, default=None)	和 get()类似, 但如果键不存在于字典中，将会添加键并将值设为 default
dict.update(dict2)	把字典 dict2 的键/值对更新到 dict 里
dict.values()	以列表返回字典中的所有值
dict.pop(key[,default])	删除字典给定键 key 所对应的值，返回值为被删除的值。key 值必须给出。 否则，返回 default 值。
dict.popitem()	随机返回并删除字典中的最后一对键和值。

实例：

```
dict1 = {'Name': 'Zara', 'Age': 7}
print("Start Len : %d" % len(dict1))
dict1.clear()
print("End Len : %d" % len(dict1))

dict1 = {'Name': 'Zara', 'Age': 7}
dict2 = dict1.copy()
print("New Dictinary:", dict2)

seq = ('name', 'age', 'sex')
dict1 = dict.fromkeys(seq)
print("New Dictionary", dict1)
dict1 = dict.fromkeys(seq, 10)
print("New Dictionary", dict1)

dict = {'Name': 'Zara', 'Age': 27}
print("Value :", dict.get('Age'))
print("Value :", dict.get('Sex', "Never"))
```

```
print("Value:", 'Age' in dict)
print("Value:", 'Sex' in dict)

dict = {'Name': 'Zara', 'Age': 7}
print("items Value :", dict.items())
print("keys Value :", dict.keys())
print("values Value :", dict.values())

print("Value :", dict.setdefault('Name', None))
print("Value :", dict.setdefault('Sex', 'male'))
print(dict)

dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female'}
dict.update(dict2)
print("Value : %s" % dict)
```

结果:

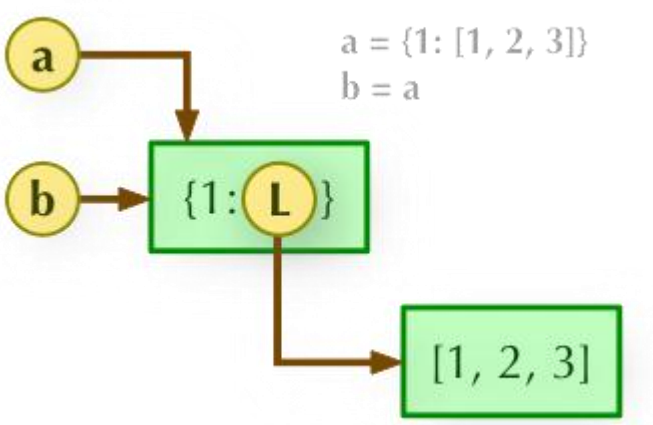
```
Start Len : 2
End Len : 0
New Dictinary: {'Name': 'Zara', 'Age': 7}
New Dictionary {'name': None, 'age': None, 'sex': None}
New Dictionary {'name': 10, 'age': 10, 'sex': 10}
Value : 27
Value : Never
Value: True
Value: False
items Value : dict_items([('Name', 'Zara'), ('Age', 7)])
keys Value : dict_keys(['Name', 'Age'])
values Value : dict_values(['Zara', 7])
Value : Zara
Value : male
{'Name': 'Zara', 'Age': 7, 'Sex': 'male'}
Value : {'Name': 'Zara', 'Age': 7, 'Sex': 'female'}
```

3.6.4. 字典生成式

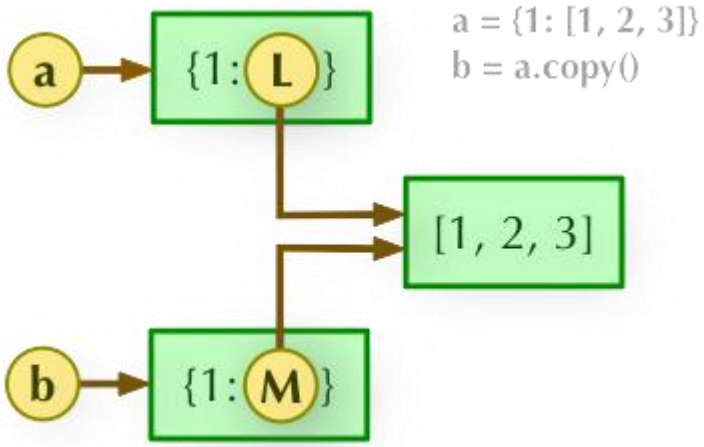
```
test_dict = {"Baidu": 1, "Google": 2, "Taobao": 3, "Zhihu": 4}
new_dict = {key: val for key, val in test_dict.items() if key != 'Zhihu'}
# 输出移除后的字典
print("字典移除后 : ", new_dict)
```

3.6.5. Python 直接赋值、浅拷贝和深度拷贝的区别

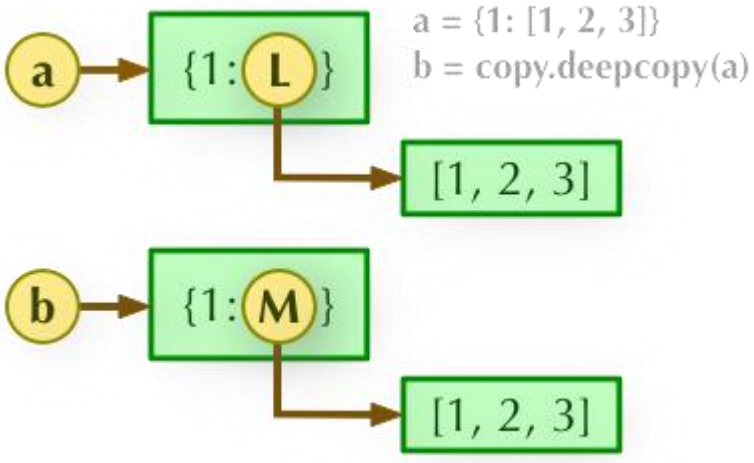
1、b = a: 赋值引用，a 和 b 都指向同一个对象。



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



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



3.7. set 无序不重复元素序列

集合（set）是一个无序的不重复元素序列。

可以使用大括号 { } 或者 set() 函数创建集合，注意：创建一个空集合必须用 set() 而不是 { }，因为 { } 是用来创建一个空字典。

创建格式：
parame = {value01,value02,...}
或者
set([iterable])
iterable -- 可迭代的对象，比如列表、字典、元组等等。

示例代码：

```
# coding=utf-8

basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)  # 这里演示的是去重功能
# {'orange', 'banana', 'pear', 'apple'}
print('orange' in basket)  # 快速判断元素是否在集合内
# True
print('crabgrass' in basket)
# False
```

3.7.1. Set 集合的交集&并集&差集

```
# 下面展示两个集合间的运算.
a = set('abracadabra')
b = set('alacazam')
print(a)
# {'a', 'r', 'b', 'c', 'd'}

print("差集:",a - b)  # 集合a 中包含而集合b 中不包含的元素

# {'r', 'd', 'b'}

print("并集:",a | b)  # 集合a 或b 中包含的所有元素

# {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

print("交集:",a & b)  # 集合a 和b 中都包含了的元素

# {'a', 'c'}

print("异或集",a ^ b)  # 不同时包含于a 和b 的元素

# {'r', 'd', 'b', 'm', 'z', 'l'}
```

3.7.2. set 的集合推导式

set 支持集合推导式(Set comprehension):

```
>>>a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a {'r', 'd'}
```

3.7.3. 集合内置方法完整列表

方法	描述
add()	将元素 x 添加到集合 s 中，如果元素已存在，则不进行任何操作。
s.discard(x)	删除集合中指定的元素，如果元素不存在，不会发生错误
s.remove(x)	将元素 x 从集合 s 中移除，如果元素不存在，则会发生错误。
clear()	移除集合中的所有元素
copy()	拷贝一个集合
difference()	返回多个集合的差集
difference_update()	移除集合中的元素，该元素在指定的集合也存在。
intersection()	返回集合的交集
intersection_update()	返回集合的交集。
isdisjoint()	判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。
issubset()	判断指定集合是否为该方法参数集合的子集。
issuperset()	判断该方法的参数集合是否为指定集合的子集
pop()	随机移除元素
symmetric_difference()	返回两个集合中不重复的元素集合。
symmetric_difference_update()	移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。
union()	返回两个集合的并集
s.update(x)	参数可以是列表，元组，字典

4. Python 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

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

4.1. 定义一个函数

规则：

1. 函数代码块以 def 关键词开头，后接函数标识符名称和圆括号()。
2. 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。
3. 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
4. 函数内容以冒号起始，并且缩进。
5. return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的 return 相当于返回

None。

语法

```
def functionname( parameters ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

示例：

```
# coding=utf-8

def printme(str):
    "打印传入的字符串到标准显示设备上"
    print(str)
    return
```

4.2. 函数的调用

示例：

```
# coding=utf-8

# 定义函数
def printme(str):
    "打印任何传入的字符串"
    print(str)
    return

# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

调用函数的时候，如果传入的参数数量不对，会报 `TypeError` 的错误，并且 Python 会明确地告诉你：abs()有且仅有 1 个参数，但给出了两个：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 `TypeError` 的错误，并且给出错误信息：str 是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: bad operand type for abs(): 'str'
```

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
>>> a = abs # 变量 a 指向 abs 函数
>>> a(-1) # 所以也可以通过 a 调用 abs 函数1
```

4.3. 定义函数

自定义一个求绝对值的 my_abs 函数：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
print(my_abs(5))
print(my_abs(-5))
```

4.3.1. return 语句

return 语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的 return 语句返回 None。

示例：

```
# 可写函数说明
def sum(arg1, arg2):
    # 返回2个参数的和."
    total = arg1 + arg2
    print("函数内 : ", total)
    return total

# 调用sum函数
total = sum(10, 20)
print("函数外 : ", total)
```

结果：

函数内 : 30
函数外 : 30

函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 `return` 语句，函数执行完毕后会返回 `None`。

`return None` 可以简写为 `return`。

4.3.2. 空函数 & pass 语句

如果想定义一个什么事也不做的空函数，可以用 `pass` 语句：

```
def nop():
    pass
```

`pass` 可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个 `pass`，让代码能运行起来。

`pass` 还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了 pass，代码运行就会有语法错误。

4.3.3. 返回多个值

```
import math
def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

这样我们就可以同时获得返回值：

```
x, y = move(100, 100, 60, math.pi / 6)
```

Python 函数返回的本质是一个元组：

```
r = move(100, 100, 60, math.pi / 6)
print(r, type(r))
```

```
(151.96152422706632, 70.0) <class 'tuple'>
```

在语法上，返回一个 tuple 可以省略括号，而多个变量可以同时接收一个 tuple，按位置赋给对应的值，所以，Python 的函数返回多值其实就是返回一个 tuple，但写起来更方便。

4.4. 可变(mutable)与不可变(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

不可变类型：

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

可变类型：

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

python 函数的参数传递：

不可变类型：

整数、字符串、元组。如 fun（a），传递的只是 a 的值，没有影响 a 对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。

可变类型：

列表，字典。如 fun（la），则是将 la 真正的传过去，修改后 fun 外部的 la 也会受影响

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

4.4.1. python 传不可变对象实例

```
# coding=utf-8
```

```
def ChangeInt(a):
    a = 10

b = 2
ChangeInt(b)
print(b)  # 结果是 2
```

实例中有 int 对象 2，指向它的变量是 b，在传递给 ChangeInt 函数时，按传值的方式复制了变量 b，a 和 b 都指向了同一个 Int 对象，在 a=10 时，则新生成一个 int 值对象 10，并让 a 指向它。

4.4.2. 传可变对象实例

```
# coding=utf-8

# 可写函数说明
def changeme(mylist):
    "修改传入的列表"
    mylist.append([1, 2, 3, 4])
    print("函数内取值: ", mylist)
    return

# 调用 changeme 函数
mylist = [10, 20, 30]
changeme(mylist)
print("函数外取值: ", mylist)
```

实例中传入函数的和在末尾添加新内容的对象用的是同一个引用，故输出结果如下：

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

4.5. 参数

以下是调用函数时可使用的正式参数类型：

- 必备参数
- 关键字参数
- 默认参数
- 不定长参数

示例 1：

```
# coding=utf-8

# 函数申明
def printinfo1(name, age=35):
    "打印任何传入的字符串"
    print("Name: ", name)
    print("Age ", age)
    return
```

```
def printinfo2(name, age, *lessons):
    "打印任何传入的字符串"
    print("Name: ", name);
    print("Age ", age)
    for lesson in lessons:
        print(lesson)
    return

# 使用关键字参数允许函数调用时参数的顺序与声明时不一致
printinfo1(age=50, name="miki")
# 缺省参数可以省略，则传入默认值
printinfo1("miki")
printinfo2("miki", 33, "python", "c++", "java", 12)
```

结果：

```
Name:  miki
Age  50
Name:  miki
Age  35
Name:  miki
Age  33
python
c++
java
12
```

上例 printinfo1 函数申明中

name 是必备参数，是必须传入的参数

age 是默认参数，可以不传入参数,=右边的是不传入参数时的默认值

调用函数时，传入的参数类型顺序必须与函数定义的参数类型顺序一致。

但使用关键字参数，则可以调用时参数的顺序与声明时不一致

上例 printinfo2 函数申明中

`lessons` 是不定长参数,不定长参数只能申明一个，而且只能申明在最后面，不定长参数也可以不传入任何参数。由于不定长参数的存在，申明默认参数则没有任何意义。

4.5.1. 默认参数

```
def enroll(name, gender, age=6, city='Beijing'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)
```

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 `enroll('Bob', 'M', 7)`，意思是，除了 `name`，`gender` 这两个参数外，最后 1 个参数应用在参数 `age` 上，`city` 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 `enroll('Adam', 'M', city='Tianjin')`，意思是，`city` 参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个 list，添加一个 `END` 再返回：

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用 add_end()时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

原因：

Python 函数在定义的时候，默认参数 L 的值就被计算出来了，即[]，因为默认参数 L 也是一个变量，它指向对象[]，每次调用该函数，如果改变了 L 的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

所以，定义默认参数要牢记一点：默认参数必须指向不可变对象！

要修改上面的例子，我们可以用 None 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计 str、None 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

4.5.2. 可变参数

可变参数就是传入的参数个数是可变的，可以是 1 个、2 个到任意个，还可以是 0 个。

我们以数学题为例子，给定一组数字 a，b，c……，请计算 $a^2 + b^2 + c^2 + ……$。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把 a，b，c……作为一个 list 或 tuple 传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
```

```
    return sum
```

但是调用的时候，需要先组装出一个 list 或 tuple：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义 list 或 tuple 参数相比，仅仅在参数前面加了一个*号。在函数内部，参数 numbers 接收到的是一个 tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括 0 个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

在 list 或 tuple 前面加一个*号，可以把 list 或 tuple 的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

4.5.3. 关键字参数

可变参数允许你传入 0 个或任意个参数，这些可变参数在函数调用时自动组装为一个 tuple。而关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个 dict。请看示例：

```
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
```

函数 person 除了必选参数 name 和 age 外，还接受关键字参数 kw。在调用该函数时，可以只传入必选参数：

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在 person 函数里，我们保证能接收到 name 和 age 这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。
和可变参数类似，也可以先组装出一个 dict，然后，把该 dict 转换为关键字参数传进去：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=kw['city'], job=kw['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **kw)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

4.5.4. 参数组合

在 Python 中定义函数，可以用必选参数、默认参数、可变参数和关键字参数，这 4 种参数都可以一起使用，或者只用其中某些，但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数和关键字参数。
比如定义一个函数，包含上述 4 种参数：

```
def func(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)
```

在函数调用的时候，Python 解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> func(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> func(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> func(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> func(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

最神奇的是通过一个 tuple 和 dict，你也可以调用该函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'x': 99}
>>> func(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'x': 99}
```

所以，对于任意函数，都可以通过类似 func(*args, **kw) 的形式调用它，无论它的参数是如何定义的。

*args 是可变参数，args 接收的是一个 tuple；
**kw 是关键字参数，kw 接收的是一个 dict。
以及调用函数时如何传入可变参数和关键字参数的语法：
可变参数既可以直接传入：func(1, 2, 3)，又可以先组装 list 或 tuple，再通过*args 传入：func(*(1, 2, 3))；
关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装 dict，再通过**kw 传入：func(**{'a': 1, 'b': 2})。
使用*args 和**kw 是 Python 的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

4.6. 匿名函数

python 使用 lambda 来创建匿名函数。

- lambda 只是一个表达式，函数体比 def 简单很多。
- lambda 的主体是一个表达式，而不是一个代码块。仅仅能在 lambda 表达式中封装有限的逻辑进去。
- lambda 函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。

语法

lambda 函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

示例：

```
# coding=utf-8

# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2

# 调用 sum 函数
print("相加后的值为 : ", sum(10, 20))
print("相加后的值为 : ", sum(20, 20))
```

结果：

```
相加后的值为 :  30
相加后的值为 :  40
```

4.7. 全局变量和局部变量

定义在函数内部的变量拥有一个局部作用域，称为局部变量，只能在其被声明的函数内部访问。
定义在函数外的拥有全局作用域，称为全局变量，可以在整个程序范围内访问。

调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。

示例：

```
# coding=utf-8

total = 0  # 这是一个全局变量

# 可写函数说明
def sum(arg1, arg2):
    # 返回 2 个参数的和."
    total = arg1 + arg2  # total 在这里是局部变量.
    print("函数内是局部变量 : ", total)
    return total

# 调用 sum 函数
sum(10, 20)
print("函数外是全局变量 : ", total)
```

结果：

```
函数内是局部变量 ：  30
函数外是全局变量 ：  0
```

如果要给全局变量在一个函数里赋值，必须使用 global 语句。
global VarName 的表达式会告诉 Python， VarName 是一个全局变量，这样 Python 就不会在局部命名空间里寻找这个变量了。
例：

```python
# coding=utf-8

Money = 2000

def AddMoney():
    global Money
    Money = Money + 1

print(Money)
AddMoney()
print(Money)
```

结果：

```
2000
2001
```

4.8. Python 常用内置函数

https://docs.python.org/zh-cn/3.7/library/functions.html

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

4.8.1. divmod() 函数

调用函数 divmod(a,b) 返回一个包含商和余数的元组(a/b, a % b)

```
# coding=utf-8

print(divmod(7, 2))
print(divmod(8, 2))
print(divmod(1 + 2j, 1 + 0.5j))
```

结果：

```
(3, 1)
(4, 0)
((1+0j), 1.5j)
```

4.8.2. enumerate() 函数

描述

enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据下标和数据，一般用在 for 循环当中。

Python 2.3. 以上版本可用，2.6 添加 start 参数。

语法

enumerate(sequence, [start=0])

参数

- sequence -- 一个序列、迭代器或其他支持迭代对象(如列表、元组或字符串)。
- start -- 下标起始位置。

```
# coding=utf-8

seasons = ['Spring', 'Summer', 'Fall', 'Winter']
print(list(enumerate(seasons)))
for i, element in enumerate(seasons):
    print(i, element)
for i, element in enumerate(seasons, 1):
    print(i, element)
```

结果：

```
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
0 Spring
1 Summer
2 Fall
3 Winter
1 Spring
2 Summer
3 Fall
4 Winter
```

4.8.3. all()与 any()函数

all() 函数用于判断给定的可迭代参数 iterable 中的所有元素是否都满足不为 0、''、False 如果所有元素都满足条件返回 True，否则返回 False。

函数等价于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

实例：

```
# coding=utf-8

print(all(['a', 'b', 'c', 'd']))  # 列表 list，元素都不为空或 0
print(all(['a', 'b', '', 'd']))  # 列表 list，存在一个为空的元素
print(all([0, 1, 2, 3]))  # 列表 list，存在一个为 0 的元素

print(all(('a', 'b', 'c', 'd')))  # 元组 tuple，元素都不为空或 0
print(all(('a', 'b', '', 'd')))  # 元组 tuple，存在一个为空的元素
print(all((0, 1, 2, 3)))  # 元组 tuple，存在一个为 0 的元素

print(all([]))  # 空列表
print(all(()))  # 空元组
```

结果：

```
True
False
False
True
False
False
True
True
```

any() 函数用于判断给定的可迭代参数 iterable 是否全部为空对象，如果都为空、0、false，则返回 False，如果不都为空、0、false，则返回 True。

函数等价于：

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

```
>>>any(['a', 'b', 'c', 'd'])  # 列表 list，元素都不为空或 0
True
>>> any(['a', 'b', '', 'd'])  # 列表 list，存在一个为空的元素
True
>>> any([0, '', False])  # 列表 list,元素全为 0,'',false
False
>>> any(('a', 'b', 'c', 'd'))  # 元组 tuple，元素都不为空或 0
```

```
True
>>> any(('a', 'b', '', 'd'))   # 元组 tuple，存在一个为空的元素
True
>>> any((0, '', False))        # 元组 tuple，元素全为 0,'',false
False
>>> any([]) # 空列表
False
>>> any(()) # 空元组
False
```

4.8.4. sorted() 函数

sorted() 函数对所有可迭代的对象进行排序操作。

sort 与 sorted 区别：

sort 是应用在 list 上的方法，sorted 可以对所有可迭代的对象进行排序操作。

list 的 sort 方法返回的是对已经存在的列表进行操作，而内建函数 sorted 方法返回的是一个新的 list，而不是在原来的基础上进行的操作。

sorted 语法：

sorted(iterable[, cmp[, key[, reverse]]])

参数说明：

- iterable -- 可迭代对象。
- cmp -- 比较的函数，这个具有两个参数，参数的值都是从可迭代对象中取出，此函数必须遵守的规则为，大于则返回 1，小于则返回-1，等于则返回 0。
- key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
- reverse -- 排序规则，reverse = True 降序 ， reverse = False 升序（默认）。

示例：

```
# coding=utf-8

a = [5, 7, 6, 3, 4, 1, 2]
b = sorted(a)  # 保留原列表
print(a,b)

L = [('b', 2), ('a', 1), ('c', 3), ('d', 4)]
print(sorted(L, key=lambda x: x[1]))  # 利用 key

students = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
print(sorted(students, key=lambda s: s[2]))  # 按年龄排序

print(sorted(students, key=lambda s: s[2], reverse=True))  # 按年龄降序
```

结果：

```
[5, 7, 6, 3, 4, 1, 2] [1, 2, 3, 4, 5, 6, 7]
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

4.8.5. map() 函数

map() 会根据提供的函数对指定序列做映射。

第一个参数 function 以参数序列中的每一个元素调用 function 函数，返回包含每次 function 函数返回值的新列表。

map() 函数语法：

map(function, iterable, ...)

参数：

- function -- 函数，有两个参数
- iterable -- 一个或多个序列

实例：

```
# coding=utf-8

def square(x):  # 计算平方数
    return x ** 2

# 所有元素变为原本的平方
print(list(map(square, [1, 2, 3, 4, 5])))
print(list(map(lambda x: x ** 2, [1, 2, 3, 4, 5])))

# 提供了两个列表，对相同位置的列表数据进行相加
print(list(map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])))
```

结果：

```
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
[3, 7, 11, 15, 19]
```

4.8.6. zip() 函数

zip() 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的迭代器，可以使用 list() 转换来输出列表。

如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同，利用 * 号操作符，可以将元组解压为列表。

zip([iterable, ...])

参数说明：

- iterabl -- 一个或多个迭代器;

实例：

```
# coding=utf-8

a = [1, 2, 3]
b = [4, 5, 6]
c = [5, 6, 7, 8, 4, 9]

zipped = zip(a, b)  # 打包为元组的列表
print(list(zipped))
print(list(zip(*zip(a, b))))
# print(list(zip()))  # 与 zip 相反，可理解为解压，返回二维矩阵式
```

```
print(list(zip(a, c)))  # 元素个数与最短的列表一致
```

结果：

```
[(1, 4), (2, 5), (3, 6)]
[(1, 2, 3), (4, 5, 6)]
[(1, 5), (2, 6), (3, 7)]
```

4.8.7. next() 函数

返回迭代器的下一个项目。

next 语法：

next(iterator[, default])

参数说明：

- iterator -- 可迭代对象
- default -- 可选，用于设置在没有下一个元素时返回该默认值，如果不设置，又没有下一个元素则会触发 StopIteration 异常。

```
# -*- coding: UTF-8 -*-

# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
        print(x,end=" ")
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

结果：

1 2 3 4 5

4.8.8. hash() 函数

hash() 用于获取取一个对象（字符串或者数值等）的哈希值。

```
# coding=utf-8

print(hash('test'))     # 字符串
print(hash(1))          # 数字
print(hash(str([1, 2, 3])))   # 集合
print(hash(str({'1': 1})))   # 字典
```

结果：

```
7016362437772442644
1
-7270873335483349735
```

3953190168926789720

4.9. 函数式编程

4.9.1. 高阶函数

高阶函数(Higher-order function)

变量可以指向函数：

```
>>> f = abs
>>> f(-10)
10
```

成功！说明变量 f 现在已经指向了 abs 函数本身。

函数名也是变量：

对于 abs() 这个函数，完全可以把函数名 abs 看成变量，它指向一个可以计算绝对值的函数！

如果把 abs 指向其他对象：

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把 abs 指向 10 后，就无法通过 abs(-10) 调用该函数了！要恢复 abs 函数，可重启 Python 交互环境。

注：由于 abs 函数实际上是定义在 __builtin__ 模块中的，所以要让修改 abs 变量的指向在其它模块也生效，要用 __builtin__.abs = 10

传入函数：

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
```

当我们调用 add(-5, 6, abs) 时，参数 x，y 和 f 分别接收 -5，6 和 abs，根据函数定义，我们可以推导计算过程为：

```
x ==> -5
y ==> 6
f ==> abs
f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
```

用代码验证一下：

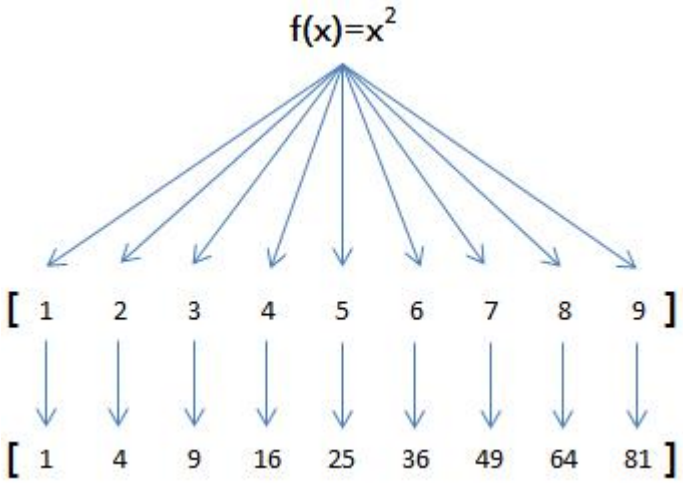
```
>>> add(-5, 6, abs)
11
```

编写高阶函数，就是让函数的参数能够接收别的函数。

4.9.2. map

`map()`函数接收两个参数，一个是函数，一个是序列，`map`将传入的函数依次作用到序列的每个元素，并把结果作为新的 list 返回。

举例说明，比如我们有一个函数 f(x)=x^2，要把这个函数作用在一个 list `[1, 2, 3, 4, 5, 6, 7, 8, 9]`上，就可以用`map()`实现如下：



直接计算的写法：

```
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print(L)
```

map 函数的写法：

```
def f(x):
    return x * x
r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
print(list(r))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

可简写为:

```
map(lambda x: x ** 2, [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`map()`传入的第一个参数是`f`，即函数对象本身。

`map()`能一眼看明白“把 f(x)作用在 list 的每一个元素并把结果生成一个新的 list”，阅读性较高。

所以，`map()`作为高阶函数，事实上它把运算规则抽象了。

利用`map()`函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：`['adam', 'LISA', 'barT']`，输出：`['Adam', 'Lisa', 'Bart']`。

```
L = ['adam', 'LISA', 'barT']
l = list(map(lambda x: x.title(), L))
print(l)
```

运行结果：

```
['Adam', 'Lisa', 'Bart']
```

4.9.3. reduce

`reduce`把一个函数作用在一个序列`[x1, x2, x3, ...]`上，这个函数必须接收两个参数，`reduce`把结果继续和序列的下一个元素做累积计算，其效果就是：

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比方说对一个序列求和，就可以用`reduce`实现：

```
from functools import reduce
def add(x, y):
    return x + y

print(reduce(add, [1, 3, 5, 7, 9]))
```

```
25
```

当然求和运算可以直接用 Python 内建函数`sum()`，没必要动用`reduce`。

但是如果要把序列`[1, 3, 5, 7, 9]`变换成整数`13579`，`reduce`就可以派上用场：

```
from functools import reduce

def fn(x, y):
    return x * 10 + y

print(reduce(fn, [1, 3, 5, 7, 9]))
```

利用 reduce()求积：

```
L = [1, 2, 3, 4, 5]
def prod(L):
    return reduce(lambda x, y: x * y, L)
print(prod(L))
```

运行结果：

120

4.9.4. filter

Python 内建的 filter()函数用于过滤序列。

和 map()类似，filter()也接收一个函数和一个序列。和 map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。

filter()函数返回的是一个 Iterator，也就是一个惰性序列，所以要强迫 filter()完成计算结果，需要用 list()函数获得所有结果并返回 list。

例如，在一个 list 中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):
    return n % 2 == 1
```

```
print(list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])))
# 结果: [1, 5, 9, 15]
```

把一个序列中的空字符串删掉，可以这么写：

```
def not_empty(s):
    return s and s.strip()

print(list(filter(not_empty, ['A', '', 'B', None, 'C', '  '])))
# 结果: ['A', 'B', 'C']
```

用 filter()筛选 1~100 的素数：

```
def isPrime(num):
    if (num < 2): return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if (num % i) == 0:
            return False
    return True


print(list(filter(isPrime, range(101))))
```

用 filter()筛选 1~1000 的素数，埃氏筛法：

```
#构造一个从 3 开始的奇数序列
def _odd_iter():
    n = 1
    while True:
        n += 2
        yield n
#定义一个筛选函数
def _not_divisible(n):
    return lambda x: x % n > 0
#定义一个生成器，不断返回下一个素数
def primes():
    yield 2
    it = _odd_iter()  # 初始序列
    while True:
        n = next(it)  # 返回序列的第一个数
        yield n
        it = filter(_not_divisible(n), it)  # 构造新序列

for n in primes():
    if n > 1000:
        break
    print(n, end=",")
```

4.9.5. sorted

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个 dict 呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素 `x` 和 `y`，如果认为 `x < y`，则返回 `-1`，如果认为 `x == y`，则返回 `0`，如果认为 `x > y`，则返回 `1`，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

Python 内置的 `sorted()` 函数就可以对 list 进行排序：

```
>>> sorted([36, 5, 12, 9, 21])
[5, 9, 12, 21, 36]
```

此外，`sorted()` 函数也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。比如，如果要倒序排序，我们就可以自定义一个 `reversed_cmp` 函数：

```
def reversed_cmp(x, y):
    if x > y:
        return -1
    if x < y:
        return 1
    return 0
```

传入自定义的比较函数 reversed_cmp，就可以实现倒序排序：

```
>>> sorted([36, 5, 12, 9, 21], reversed_cmp)
[36, 21, 12, 9, 5]
```

我们再看一个字符串排序的例子：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'])
['Credit', 'Zoo', 'about', 'bob']
```

默认情况下，对字符串排序，是按照 ASCII 的大小比较的，由于'Z' < 'a'，结果，大写字母 Z 会排在小写字母 a 的前面。

忽略大小写，按照字母序排序：

```
print(sorted(['bob', 'about', 'Zoo', 'Credit'], key=lambda a: a.upper()))
```

4.9.6. 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数！

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function sum at 0x10452f668>
```

调用函数 f 时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，我们在函数 lazy_sum 中又定义了函数 sum，并且，内部函数 sum 可以引用外部函数 lazy_sum 的参数和局部变量，当 lazy_sum 返回函数 sum 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 lazy_sum() 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

f1() 和 f2() 的调用结果互不影响。

4.9.7. 闭包

注意到返回的函数在其定义内部引用了局部变量 args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 f() 才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
             return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的 3 个函数都返回了。

你可能认为调用 f1()，f2() 和 f3() 结果应该是 1，4，9，但实际结果是：

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是 9！原因就在于返回的函数引用了变量 i，但它并非立刻执行。等到 3 个函数都返回时，它们所引用的变量 i 已经变成了 3，因此最终结果为 9。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def count():
    fs = []
    for i in range(1, 4):
        def f(j):
```

```
            def g():
                return j*j
            return g
        fs.append(f(i))
    return fs
```

```
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

缺点是代码较长，可利用 lambda 表达式缩短代码:

```
def count():
    fs = []
    for i in range(1, 4):
        f = lambda j: (lambda: j * j)
        fs.append(f(i))
    return fs

f1, f2, f3 = count()
print(f1(), f2(), f3())
```

运行结果:

```
1 4 9
```

4.9.8. 装饰器

完整示例:

```
import functools

def log(text):

    def decorator(func):

        @functools.wraps(func)

        def wrapper(*args, **kw):

            print('%s %s():' % (text, func.__name__))

            return func(*args, **kw)

        return wrapper

    return decorator
```

函数对象有一个__name__属性，可以拿到函数的名字：

```
>>> def now():
...     print('2015-3-25')
...
>>> now.__name__
'now'
```

现在，假设我们要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator 就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的 decorator，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

wrapper()函数的参数定义是(*args, **kw)，因此，wrapper()函数可以接受任意参数的调用。

观察上面的log，因为它是一个 decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助 Python 的@语法，把 decorator 置于函数的定义处：

```
@log
def now():
    print('2015-3-25')
```

把@log放到now()函数的定义处，相当于执行了语句：

```
now = log(now)
```

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

```
>>> now()
call now():
2015-3-25
```

如果 decorator 本身需要传入参数，那就需要编写一个返回 decorator 的高阶函数，写出来会更复杂。比如，要自定义 log 的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个 3 层嵌套的 decorator 用法如下：

```
@log('execute')
def now():
    print('2015-3-25')
```

和两层嵌套的 decorator 相比，3 层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

执行结果如下：

```
>>> now()
execute now():
2015-3-25
```

以上两种 decorator 的定义都没有问题，但经过 decorator 装饰之后的函数，它们的`__name__`已经从原来的`'now'`变成了`'wrapper'`：

```
>>> now.__name__
'wrapper'
```

Python 内置的`functools.wraps`能达到`wrapper.__name__ = func.__name__`的效果，所以，一个完整的 decorator 的写法如下：

```
import functools
def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
```

```
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的 decorator：

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

练习

设计一个 decorator，它可作用于任何函数上，并打印该函数的执行时间，可以计算任何函数执行时间：

```
import functools, time

def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kw):
        start_time = time.time() * 1000
        result = fn(*args, **kw)
        run_time = time.time() * 1000 - start_time
        print('%s executed in %s ms' % (fn.__name__, run_time))
        return result
    return wrapper

@metric
def fast(x, y):
    time.sleep(0.003)
    return x + y
@metric
def slow(x, y, z):
    time.sleep(0.1257)
    return x * y * z
f = fast(11, 22)
s = slow(11, 22, 33)
```

运行结果：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
fast executed in 3.000244140625 ms
slow executed in 126.0068359375 ms
```

4.9.9. 偏函数

假设要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，于是，我们想到，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

`functools.partial` 可以创建一个偏函数，不需要我们自己定义 `int2()`，可以直接使用下面的代码创建一个新的函数 `int2`：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

创建偏函数时，实际上可以接收函数对象、`*args` 和 `**kw` 这 3 个参数，当传入：

```
int2 = functools.partial(int, base=2)
```

实际上固定了 int()函数的关键字参数 `base`，也就是：

```
int2('10010')
```

相当于：

```
kw = { 'base': 2 }
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把 `10` 作为`*args`的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)

max(*args)
```

结果为 `10`。

4.10. Python 模块

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。

下面是一个使用 python 标准库中模块的例子。

```
# -*- coding: utf-8 -*-

import sys
print('\n'.join(sys.path))
```

运行结果：

```
(base) D:\PycharmProjects\demo1>python pythonlearn/test.py
D:\PycharmProjects\demo1\pythonlearn
D:\Anaconda3\python37.zip
D:\Anaconda3\DLLs
D:\Anaconda3\lib
D:\Anaconda3
D:\Anaconda3\lib\site-packages
D:\Anaconda3\lib\site-packages\win32
D:\Anaconda3\lib\site-packages\win32\lib
D:\Anaconda3\lib\site-packages\Pythonwin
```

可修改 PYTHONPATH 环境变量，增加模块搜索路径：

```
(base) D:\PycharmProjects\demo1>set PYTHONPATH=.


(base) D:\PycharmProjects\demo1>python pythonlearn/test.py
D:\PycharmProjects\demo1\pythonlearn
D:\PycharmProjects\demo1
```

```
D:\Anaconda3\python37.zip
D:\Anaconda3\DLLs
D:\Anaconda3\lib
D:\Anaconda3
D:\Anaconda3\lib\site-packages
D:\Anaconda3\lib\site-packages\win32
D:\Anaconda3\lib\site-packages\win32\lib
D:\Anaconda3\lib\site-packages\Pythonwin
```

4.10.1. import 语句

语法如下：

import module1[, module2[,... moduleN]

一个模块只会被导入一次，不管你执行了多少次 import。这样可以防止导入模块被一遍又一遍地执行。

当我们使用 import 语句的时候，Python 解释器搜索路径中查找，搜索路径是由一系列目录名组成的，Python 解释器就依次从这些目录中去寻找所引入的模块。

test.py 代码示例：

```
# coding=utf-8

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("python")
```

结果：

Hello : python

一个模块只会被导入一次，不管你执行了多少次 import

4.10.2. From…import 语句

Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中。

From…import 相当于 java 的静态导入，只导入一个模块的指定部分

语法如下：

from modname import name1[, name2[, ... nameN]]

例如，要导入模块 `string` 的 `maketrans` 函数，使用如下语句：

```
from string import maketrans   # 必须调用 maketrans 函数。
```

```
intab = "aeiou"
outtab = "12345"
#定义一间转换表
trantab = maketrans(intab, outtab)
```

和 import 的区别

```
import string
intab = "aeiou"
outtab = "12345"
string.maketrans(intab, outtab)
```

从上面的示例可以看出，From…import 导入的方法可以直接调用，import 导入的方法必须加上类名

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：
from modname import *

例如我们想一次性引入 math 模块中所有的东西，语句如下：
from math import *

4.10.3. 模块的搜索路径

默认情况下，Python 解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在 `sys` 模块的 `path` 变量中：

```
>>> import sys
>>> sys.path
['', '/Library/Python/2.7/site-packages/pycrypto-2.6.1-py2.7-macosx-10.9-intel.egg', '/Library/Python/2.7/site-packages/PIL-1.1.7-py2.7-macosx-10.9-intel.egg', ...]
```

如果我们要添加自己的搜索目录，有两种方法：
一是直接修改 `sys.path`，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('E:/demo')
```

这种方法是在运行时修改，运行结束后失效。
第二种方法是设置环境变量 `PYTHONPATH`，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置 Path 环境变量类似。注意只需要添加你自己的搜索路径，Python 自己本身的搜索路径不受影响。
例如：
set PYTHONPATH=.

4.10.4. dir()找到模块内定义的所有名称

dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回：

```
# coding=utf-8
```

```
import math
print(dir(math))
```

结果：

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

在这里，特殊字符串变量__name__指向模块的名字，__file__指向该模块的导入文件名。

如果没有给定参数，那么 dir() 函数会罗列出当前定义的所有名称：
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'math']

4.10.5. globals() 和 locals() 函数

根据调用地方的不同，globals() 和 locals() 函数可被用来返回全局和局部命名空间里的名字。
如果在函数内部调用 locals()，返回的是所有能在该函数里访问的命名。
如果在函数内部调用 globals()，返回的是所有在该函数里能访问的全局名字。
两个函数的返回类型都是字典。所以名字们能用 keys() 函数摘取。

4.10.6. __all__&从一个包中导入*

from sound.effects import *
会执行 effects 包下面的__init__.py 并导入里面定义的内容，如果存在 __all__ 列表变量，那么就可以导入该列表指定的模块。
在:file:sounds/effects/__init__.py 中包含如下代码:

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用 from sound.effects import *这种用法时，会导入包里面这三个子模块。

4.10.7. __main__

Python 解释器会将运行入口模块文件的__name__置为__main__

```
if __name__=='__main__':

    test()
```

当我们在命令行运行 `hello` 模块文件时，Python 解释器把一个特殊变量`__name__`置为`__main__`，而如果在其他地方导入该 `hello` 模块时，`if` 判断将失败，因此，这种 `if` 测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

5. 文件 I/O

5.1. 基本方法

5.1.1. 打开和关闭文件

用 open()函数打开一个文件，创建一个 file 对象，调用相关的方法即可对文件进行读写。
完整的语法格式为：

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

参数说明:
file: 必需，文件路径（相对或者绝对路径）。

- mode: 可选，文件打开模式
- buffering: 寄存区的缓冲大小。设为 0，不会有寄存。设为负值，寄存区的缓冲大小则为系统默认。默认寄存区的缓冲大小为 0
- encoding: 一般使用 utf8
- errors: 报错级别
- newline: 区分换行符
- closefd: 传入的 file 参数类型
- opener:

access_mode 具体参数列表：

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。

a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

总结：

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

5.1.2. File 对象的属性

以下是和 file 对象相关的所有属性的列表：

属性	描述
file.closed	返回 true 如果文件已被关闭，否则返回 false。
file.mode	返回被打开文件的访问模式。
file.name	返回文件的名称。
file.softspace	如果用 print 输出后，必须跟一个空格符，则返回 false。否则返回 true。

File 对象的 close（）方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。
当一个文件对象的引用被重新指定给另一个文件时，Python 会关闭之前的文件。用 close（）方法关闭文件是一个很好的习惯。

5.1.3. 文件基本读写与定位

write()方法可将任何字符串写入一个打开的文件。
语法：

fileObject.write(string);

read()方法从一个打开的文件中读取指定长度的字符串。

语法：

fileObject.read([count]);

tell()描述文件当前位置

fileObject.tell()

seek()改变当前文件的位置

seek(offset [,from])

- Offset 变量表示要移动的字节数
- From 变量指定参考位置。0，表示开头。1，表示当前的位置。2，表示末尾(默认为 0)

例如：

seek(x,0) ： 从起始位置即文件首行首字符开始移动 x 个字符

seek(x,1) ： 表示从当前位置往后移动 x 个字符

seek(-x,2)：表示从文件的结尾往前移动 x 个字符

```
# coding=utf-8

# 以二进制格式打开一个文件用于读写.如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
fo = open("foo.txt", "wb+")
# 向文件写入 2 句话
fo.write(b"blog.xiaoxiaoming.xyz!\nVery good site!\n")
seq = [b"blog.xiaoxiaoming.xyz!\n", b"Very good site!"]
fo.writelines(seq)

# 查找当前位置

print("当前文件位置 : ", fo.tell())

# 写完数据，文件指针在文件末尾，现在重定向到文件开头
fo.seek(0)
# 读取 10 个字符

print("当前文件位置 : ", fo.tell())

str = fo.read(10)

print("读取的字符串是 : ", str)

print("当前文件位置 : ", fo.tell())

# 关闭打开的文件
fo.close()
```

结果：

```
当前文件位置 ： 63
当前文件位置 ： 0
读取的字符串是 ： b’www.edu360’
当前文件位置 ： 10
```

foo.txt 内容：

```
blog.xiaoxiaoming.xyz!
Very good site!
blog.xiaoxiaoming.xyz!
Very good site!
```

5.1.4. f.close()

处理完一个文件后，调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

当处理一个文件对象时，使用 with 关键字是非常好的方式。在结束后，它会帮你正确的关闭文件。 而且写起来也比 try - finally 语句块要简短:

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

5.1.5. File(文件)的方法

file 对象使用 open 函数来创建，下表列出了 file 对象常用的函数：

方法	描述
file.close()	关闭文件。关闭后文件不能再进行读写操作。
file.flush()	刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件，而不是被动的等待输出缓冲区写入。
file.read([size])	从文件读取指定的字节数，如果未给定或为负则读取所有。
file.readline([size])	读取整行，包括 "\n" 字符。
file.readlines([sizehint])	读取所有行并返回列表，若给定 sizeint>0，返回总和大约为 sizeint 字节的行，实际读取值可能比 sizhint 较大，因为需要填充缓冲区。
file.seek(offset[, whence])	设置文件当前位置
file.tell()	返回文件当前位置。
file.write(str)	将字符串写入文件，没有返回值。
file.writelines(sequence)	向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

调用 read()会一次性读取文件的全部内容，如果文件有 10G，内存就爆了，所以，要保险起见，可以反复调用 read(size)方法，每次最多读取 size 个字节的内容。

调用 next()或 readline()可以每次读取一行内容，f.next() 逐行读取数据，和 f.readline()相似，唯一不同的是，**readline() 读取到最后如果没有数据会返回空，而 next() 没读取到数据则会报错。**

调用 readlines()一次读取所有内容并按行返回 list。因此，要根据需要决定怎么调用。

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

实例：

创建 tmp.txt 文件内容为：

```
这是第一行
这是第二行
这是第三行
这是第四行
这是第五行
```

代码：

```
# coding=utf-8

fo = open("tmp.txt", "r", encoding="utf-8")
# 刷新缓冲区
fo.flush()

print("文件名为: ", fo.name)

print("文件描述符为: ", fo.fileno())

print("文件连接到一个终端设备:", fo.isatty())

# next 返回文件下一行
for index in range(5):
    line = fo.readline()

    print("第 %d 行 - %s" % (index + 1, line), end="")

print()

fo.seek(0)
for line in fo.readlines():
    print(line, end="")

# 关闭文件
fo.close()
```

结果：

```
文件名为:  xiaoniu.txt
文件描述符为:  3
文件连接到一个终端设备: False
第 1 行 - 这是第一行
第 2 行 - 这是第二行
第 3 行 - 这是第三行
第 4 行 - 这是第四行
第 5 行 - 这是第五行
这是第一行
这是第二行
这是第三行
这是第四行
这是第五行
```

5.1.6. 标准的按行读取文件的方法

每次都写 try ... finally 来关闭文件实在太繁琐，所以，Python 引入了 with 语句来自动帮我们调用 close()方法，代码如下:

```
with open("tmp.txt", encoding="utf-8") as f:
    for line in f:
        print(line, end="")
```

5.2. 异常处理

```
try:

    正常的操作

except Exception1[, Exception2[,...ExceptionN]]][,Argument]as err:

    发生异常，执行这块代码

else:

    如果没有异常执行这块代码

finally:

    退出 try 时总会执行执行这块代码
```

一个 try 语句可能包含多个 except 子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。
一个 except 子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如:

```
except (RuntimeError, TypeError, NameError):
    pass
```

最后一个 except 子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

```
import sys

try:
    f = open('tmp.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
```

```
    raise
```

代码：

```
# coding=utf-8

try:
    fh = open("testfile", "r")
    try:

        fh.write("这是一个测试文件，用于测试异常!!")

    finally:

        print("关闭文件")

        fh.close()
except IOError as value:

    print("Error: 没有找到文件或读取文件失败:", value)
```

5.2.1. 抛出异常

Python 使用 raise 语句抛出一个指定的异常。例如:

```
>>>raise NameError('HiThere')
 Traceback (most recent call last):
   File "<stdin>", line 1, in ?
 NameError: HiThere
```

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise

An exception flew by!
Traceback(most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

5.2.2. 自定义异常

异常类继承自 Exception 类，可以直接继承，或者间接继承，例如:

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return str(self.value)

try:
    raise MyError(2 * 2)
except MyError as e:
    print('My exception occurred, value:', e.value)
```

```
My exception occurred, value: 4
```

```
raise MyError('oops!')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类:

```
class Error(Exception):
    pass

class InputError(Error):
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

大多数的异常的名字都以"Error"结尾，就跟标准的异常命名一样。

5.2.3. python 异常体系

https://docs.python.org/zh-cn/3.7/library/exceptions.html#exception-hierarchy

```
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
```

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于 Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)

UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

5.3. StringIO 和 BytesIO

5.3.1. StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。
StringIO 顾名思义就是在内存中读写 str。

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

```
from io import StringIO

# write to StringIO:
f = StringIO()
f.write('hello')
f.write(' ')
f.write('world!')
print(f.getvalue())
```

getvalue()方法用于获得写入后的 str。

要读取 StringIO，可以用一个 str 初始化 StringIO，然后，像读文件一样读取：

```
f = StringIO('水面细风生，\n菱歌慢慢声。\n客亭临小市，\n灯火夜妆明。')

for line in f:
    print(line.strip())
```

5.3.2. BytesIO

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

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

```
from io import BytesIO

# write to BytesIO:
f = BytesIO()
f.write(b'hello')
f.write(b' ')
f.write(b'world!')
print(f.getvalue())
```

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

```
data = '人闲桂花落，夜静春山空。月出惊山鸟，时鸣春涧中。'.encode('utf-8')

f = BytesIO(data)
print(f.read())
```

5.4. pickle 序列化

python 的 pickle 模块实现了基本的数据序列和反序列化。

通过 pickle 模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过 pickle 模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

```
pickle.dump(obj, file, [,protocol])
```

有了 pickle 这个对象，就能对 file 以读取的形式打开:

```
x = pickle.load(file)
```

注解：从 file 中读取一个字符串，并将它重构为原来的 python 对象。

file: 类文件对象，有 read()和 readline()接口。

```
# coding=utf-8

import pickle,pprint

# 使用pickle 模块将数据对象保存到文件

data1 = {'a': [1, 2.0, 3, 4 + 6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

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

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

output.close()
```

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

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

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

pkl_file.close()
```

5.5. os 模块方法

os 模块提供了非常丰富的方法用来处理文件和目录。

常用方法示例：

```
#coding=utf-8

import os
```

```
# 重命名文件 test1.txt 到 test2.txt。
os.rename( "test1.txt", "test2.txt" )
# 删除一个已经存在的文件 test2.txt
os.remove("test2.txt")
# 创建目录 test
os.mkdir("test")
```

```
#coding=utf-8

import os

# 给出当前的目录
print(os.getcwd())
#显示当前目录的所有文件名
print(os.listdir(os.getcwd()))
# 将当前目录改为"/home/newdir"
os.chdir("/hdfs/txt")
# 给出当前的目录
print(os.getcwd())
```

```
#coding=utf-8

import os

# rmdir()方法删除目录，目录名称以参数传递。
# 在删除这个目录之前，它的所有内容应该先被清除。
os.rmdir("/tmp/hive")
```

如果/tmp/hive 目录里有内容，则会抛出异常

```
Traceback (most recent call last):
  File "***/os.py", line 7, in <module>
    os.rmdir("/tmp/hive")
WindowsError: [Error 145] : '/tmp/hive'
```

5.5.1. os 递归删除目录

```
# coding=utf-8

def rmDirs(dirPath):
    import os
    for root, dirs, files in os.walk(dirPath, topdown=False):
        for name in files:
            os.remove(os.path.join(root, name))
        for name in dirs:
            os.rmdir(os.path.join(root, name))
    os.rmdir(dirPath)
rmDirs("/tmp/hive")
```

5.5.2. os 常用的方法：

方法	描述
os.chdir(path)	改变当前工作目录
os.chmod(path, mode)	更改权限
os.chown(path, uid, gid)	更改文件所有者
os.chroot(path)	改变当前进程的根目录
os.getcwd()	返回当前工作目录
os.getcwdu()	返回一个当前工作目录的 Unicode 对象
os.lchflags(path, flags)	设置路径的标记为数字标记，类似 chflags()，但是没有软链接
os.lchmod(path, mode)	修改连接文件权限
os.lchown(path, uid, gid)	更改文件所有者，类似 chown，但是不追踪链接。
os.link(src, dst)	创建硬链接，名为参数 dst，指向参数 src
os.listdir(path)	返回 path 指定的文件夹包含的文件或文件夹的名字的列表。
os.lstat(path)	像 stat(),但是没有软链接
os.makedirs(path[, mode])	递归文件夹创建函数。像 mkdir()，但创建的所有 intermediate-level 文件夹需要包含子文件夹。
os.mkdir(path[, mode])	以数字 mode 的 mode 创建一个名为 path 的文件夹.默认的 mode 是 0777 (八进制)。
os.pathconf(path, name)	返回相关文件的系统配置信息。
os.readlink(path)	返回软链接所指向的文件
os.remove(path)	删除路径为 path 的文件。如果 path 是一个文件夹，将抛出 OSError；查看下面的 rmdir()删除一个 directory。
os.removedirs(path)	递归删除目录。
os.rename(src, dst)	重命名文件或目录，从 src 到 dst
os.renames(old, new)	递归地对目录进行更名，也可以对文件进行更名。
os.rmdir(path)	删除 path 指定的空目录，如果目录非空，则抛出一个 OSError 异常。
os.statvfs(path)	获取指定路径的文件系统统计信息
os.symlink(src, dst)	创建一个软链接
os.unlink(path)	删除文件路径
os.utime(path, times)	返回指定的 path 文件的访问和修改的时间。
os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])	输出在文件夹中的文件名通过在树中游走，向上或者向下。

5.5.3. os.walk()示例

os.walk() 方法用于通过在目录树种游走输出在目录中的文件名，向上或者向下。

walk()方法语法格式如下：

```
os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])
```

参数：

- **top** -- 根目录下的每一个文件夹(包含它自己)，产生 3-元组 (dirpath，dirnames，filenames)【文件夹路径，文件夹名字，文件名】。
- **topdown** --可选，为 True 或者没有指定，一个目录的的 3-元组将比它的任何子文件夹的 3-元组先产生 (目录自上而下)。如果 topdown 为 False，一个目录的 3-元组将比它的任何子文件夹的 3-元组后产生 (目录自下而上)。
- **onerror** -- 可选，是一个函数；它调用时有一个参数，一个 OSError 实例。报告这错误后，继续 walk,或者抛出 exception 终止 walk。
- **followlinks** -- 设置为 true，则通过软链接访问目录。

示例：

```
import os

for root, dirs, files in os.walk("."):
    for name in files:
        print(os.path.join(root, name))
    for name in dirs:
        print(os.path.join(root, name))
```

5.5.4. 树形显示目录

```
import os
def show_dir(dir, layer=0):
    listdir = os.listdir(dir)
    for index, file in enumerate(listdir):
        file_path = os.path.join(dir, file)
        print("|  " * (layer - 1), end="")
        if (layer > 0):
            print("`--" if index == len(listdir) - 1 else "|--", end="")
        print(file)
        if (os.path.isdir(file_path)):
            show_dir(file_path, layer + 1)
show_dir(r"D:\PycharmProjects\awesome-webapp\www")
```

```
static
|--css
|   |--addons
|   |   |--uikit.addons.min.css
|   |   |--uikit.almost-flat.addons.min.css
|   |   `--uikit.gradient.addons.min.css
|   |--awesome.css
|   |--uikit.almost-flat.min.css
|   |--uikit.gradient.min.css
```

```
|   `--uikit.min.css
|--fonts
|   |--fontawesome-webfont.eot
|   |--fontawesome-webfont.ttf
|   |--fontawesome-webfont.woff
|   `--FontAwesome.otf
|--img
|   `--user.png
|--js
|   |--awesome.js
|   |--jquery.min.js
|   |--sha1.min.js
|   |--sticky.min.js
|   |--uikit.min.js
|   `--vue.min.js
`--README
templates
|--blog.html
|--blogs.html
|--manage_blogs.html
|--manage_blog_edit.html
|--manage_comments.html
|--manage_users.html
|--register.html
|--signin.html
`--__base__.html
```

5.5.5. os.path 常用方法

方法	说明
os.path.abspath(path)	返回绝对路径
os.path.basename(path)	返回文件名
os.path.commonprefix(list)	返回 list(多个路径)中，所有 path 共有的最长的路径
os.path.dirname(path)	返回文件路径
os.path.exists(path)	路径存在则返回 True,路径损坏返回 False
os.path.lexists	路径存在则返回 True,路径损坏也返回 True
os.path.expanduser(path)	把 path 中包含的"~"和"~user"转换成用户目录
os.path.expandvars(path)	根据环境变量的值替换 path 中包含的"$name"和"${name}"
os.path.getatime(path)	返回最近访问时间（浮点型秒数）
os.path.getmtime(path)	返回最近文件修改时间
os.path.getctime(path)	返回文件 path 创建时间
os.path.getsize(path)	返回文件大小，如果文件不存在就返回错误
os.path.isabs(path)	判断是否为绝对路径
os.path.isfile(path)	判断路径是否为文件
os.path.isdir(path)	判断路径是否为目录
os.path.islink(path)	判断路径是否为链接

os.path.ismount(path)	判断路径是否为挂载点
os.path.join(path1[, path2[, ...]])	把目录和文件名合成一个路径
os.path.normcase(path)	转换 path 的大小写和斜杠
os.path.normpath(path)	规范 path 字符串形式
os.path.realpath(path)	返回 path 的真实路径
os.path.relpath(path[, start])	从 start 开始计算相对路径
os.path.samefile(path1, path2)	判断目录或文件是否相同
os.path.sameopenfile(fp1, fp2)	判断 fp1 和 fp2 是否指向同一文件
os.path.samestat(stat1, stat2)	判断 stat tuple stat1 和 stat2 是否指向同一个文件
os.path.split(path)	把路径分割成 dirname 和 basename，返回一个元组
os.path.splitdrive(path)	一般用在 windows 下，返回驱动器名和路径组成的元组
os.path.splitext(path)	分割路径，返回路径名和文件扩展名的元组
os.path.splitunc(path)	把路径分割为加载点与文件
os.path.walk(path, visit, arg)	遍历 path，进入每个目录都调用 visit 函数，visit 函数必须有 3 个参数(arg, dirname, names)，dirname 表示当前目录的目录名，names 代表当前目录下的所有文件名，args 则为 walk 的第三个参数
os.path.supports_unicode_filenames	设置是否支持 unicode 路径名

示例：

```
import os

print(os.path.basename('/root/test/tmp.txt'))  # 返回文件名
print(os.path.dirname('/root/test/tmp.txt'))  # 返回目录路径
print(os.path.split('/root/test/tmp.txt'))  # 分割文件名与路径
print(os.path.splitext('/path/to/file.txt')) # 拆分文件扩展名
print(os.path.join('root', 'aaa', 'tmp.txt'))  # 将目录和文件名合成一个路径
```

```
tmp.txt
/root/test
('/root/test', 'tmp.txt')
('/path/to/file', '.txt')
root\aaa\tmp.txt
```

输出文件的相关信息：

```
import os,time

file = 'tmp.txt'  # 文件路径

print(os.path.getatime(file))  # 输出最近访问时间
print(os.path.getctime(file))  # 输出文件创建时间
print(os.path.getmtime(file))  # 输出最近修改时间
print(time.gmtime(os.path.getmtime(file)))  # 以 struct_time 形式输出最近修改时间
print(os.path.getsize(file))  # 输出文件大小（字节为单位）
print(os.path.abspath(file))  # 输出绝对路径
print(os.path.normpath(file))  # 规范 path 字符串形式
```

```
1569839225.44611
1569839199.8376453
```

```
1569839225.4471102
time.struct_time(tm_year=2019, tm_mon=9, tm_mday=30, tm_hour=10, tm_min=27, tm_sec=5, tm_wday=0, tm_yday=273, tm_isdst=0)
83
D:\PycharmProjects\demo1\pythonlearn\tmp.txt
tmp.txt
```

5.6. Shutil 的常用用法

shutil 模块提供了许多关于文件和文件集合的高级操作，特别提供了支持文件复制和删除的功能。
shutil.rmtree 递归删除目录
shutil.copytree 递归复制目录
shutil.copy 复制文件

```
dir_js = os.path.join(l1dir, "js")
dir_css = os.path.join(l1dir, "css")
if (os.path.exists(dir_css)):
    shutil.rmtree(dir_css)
if (os.path.exists(dir_js)):
    shutil.rmtree(dir_js)
shutil.copytree("htmlModule/JS_zoomsize/css", dir_css)
shutil.copytree("htmlModule/JS_zoomsize/js", dir_js)
shutil.copy("htmlModule/css/base.css", os.path.join(dir_css, "base.css"))
shutil.copy("htmlModule/js/media_enhance.js", os.path.join(dir_js, "media_enhance.js"))
shutil.copy("htmlModule/js/base.js", os.path.join(dir_js, "base.js"))
```

5.6.1. 文件夹与文件操作

copyfileobj(fsrc, fdst, length=16*1024)： 将 fsrc 文件内容复制至 fdst 文件，length 为 fsrc 每次读取的长度，用做缓冲区大小

- fsrc： 源文件
- fdst： 复制至 fdst 文件
- length： 缓冲区大小，即 fsrc 每次读取的长度

copyfile(src, dst)： 将 src 文件内容复制至 dst 文件

- src： 源文件路径
- dst： 复制至 dst 文件，若 dst 文件不存在，将会生成一个 dst 文件；若存在将会被覆盖
- follow_symlinks：设置为 True 时，若 src 为软连接，则当成文件复制；如果设置为 False，复制软连接。默认为 True。Python3 新增参数

copymode(src, dst)： 将 src 文件权限复制至 dst 文件。文件内容，所有者和组不受影响

- src： 源文件路径
- dst： 将权限复制至 dst 文件，dst 路径必须是真实的路径，并且文件必须存在，否则将会报文件找不到错误

- follow_symlinks：设置为 False 时，src, dst 皆为软连接，可以复制软连接权限，如果设置为 True，则当成普通文件复制权限。默认为 True。Python3 新增参数

copystat(src, dst)： 将权限，上次访问时间，上次修改时间以及 src 的标志复制到 dst。文件内容，所有者和组不受影响

- src： 源文件路径
- dst： 将权限复制至 dst 文件，dst 路径必须是真实的路径，并且文件必须存在，否则将会报文件找不到错误
- follow_symlinks：设置为 False 时，src, dst 皆为软连接，可以复制软连接权限、上次访问时间，上次修改时间以及 src 的标志，如果设置为 True，则当成普通文件复制权限。默认为 True。Python3 新增参数

copy(src, dst)： 将文件 src 复制至 dst。dst 可以是个目录，会在该目录下创建与 src 同名的文件，若该目录下存在同名文件，将会报错提示已经存在同名文件。权限会被一并复制。本质是先后调用了 copyfile 与 copymode 而已

- src：源文件路径
- dst：复制至 dst 文件夹或文件
- follow_symlinks：设置为 False 时，src, dst 皆为软连接，可以复制软连接权限，如果设置为 True，则当成普通文件复制权限。默认为 True。Python3 新增参数

copy2(src, dst)： 将文件 src 复制至 dst。dst 可以是个目录，会在该目录下创建与 src 同名的文件，若该目录下存在同名文件，将会报错提示已经存在同名文件。权限、上次访问时间、上次修改时间和 src 的标志会一并复制至 dst。本质是先后调用了 copyfile 与 copystat 方法而已

- src：源文件路径
- dst：复制至 dst 文件夹或文件
- follow_symlinks：设置为 False 时，src, dst 皆为软连接，可以复制软连接权限、上次访问时间，上次修改时间以及 src 的标志，如果设置为 True，则当成普通文件复制权限。默认为 True。Python3 新增参数

ignore_patterns(*patterns)： 忽略模式，用于配合 `copytree()` 方法，传递文件将会被忽略，不会被拷贝

- patterns：文件名称，元组

copytree(src, dst, symlinks=False, ignore=None)： 拷贝文档树，将 src 文件夹里的所有内容拷贝至 dst 文件夹

- src：源文件夹
- dst：复制至 dst 文件夹，该文件夹会自动创建，需保证此文件夹不存在，否则将报错
- symlinks：是否复制软连接，True 复制软连接，False 不复制，软连接会被当成文件复制过来，默认 False
- ignore：忽略模式，可传入 `ignore_patterns()`
- copy_function：拷贝文件的方式，可以传入一个可执行的处理函数，默认为 copy2，Python3 新增参数
- ignore_dangling_symlinks：sysmlinks 设置为 False 时，拷贝指向文件已删除的软连接时，将会报错，如果想消除这个异常，可以设置此值为 True。默认为 False,Python3 新增参数

rmtree(path, ignore_errors=False, onerror=None)： 移除文档树，将文件夹目录删除

- ignore_errors：是否忽略错误，默认 False
- onerror：定义错误处理函数，需传递一个可执行的处理函数，该处理函数接收三个参数：函数、路径和 excinfo

move(src, dst)： 将 src 移动至 dst 目录下。若 dst 目录不存在，则效果等同于 src 改名为 dst。若 dst 目录存在，将会把 src 文件夹的所有内容移动至该目录下面

- src：源文件夹或文件

- dst：移动至 dst 文件夹，或将文件改名为 dst 文件。如果 src 为文件夹，而 dst 为文件将会报错
- copy_function：拷贝文件的方式，可以传入一个可执行的处理函数。默认为 copy2，Python3 新增参数

disk_usage(path)： 获取当前目录所在硬盘使用情况。Python3 新增方法

- path：文件夹或文件路径。windows 中必须是文件夹路径，在 linux 中可以是文件路径和文件夹路径

chown(path, user=None, group=None)： 修改路径指向的文件或文件夹的所有者或分组。Python3 新增方法

- path：路径
- user：所有者，传递 user 的值必须是真实的，否则将报错 no such user
- group：分组，传递 group 的值必须是真实的，否则将报错 no such group

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

5.6.2. 归档操作

shutil 还提供了创建和读取压缩和存档文件的高级使用程序。内部实现主要依靠的是 zipfile 和 tarfile 模块

make_archive(base_name, format, root_dir, ...)： 生成压缩文件

- base_name：压缩文件的文件名，不允许有扩展名，因为会根据压缩格式生成相应的扩展名
- format：压缩格式
- root_dir：将制定文件夹进行压缩

```
import shutil,os
base_name = os.path.join(os.getcwd(),"aaa")
format = "zip"
root_dir = r"D:\PycharmProjects\demo1\numpytest"
# 将会 root_dir 文件夹下的内容进行压缩，生成一个 aaa.zip 文件
shutil.make_archive(base_name, format, root_dir)
```

get_archive_formats()： 获取支持的压缩文件格式。目前支持的有：tar、zip、gztar、bztar。在 Python3 还多支持一种格式 xztar

unpack_archive(filename, extract_dir=None, format=None)： 解压操作。Python3 新增方法

- filename：文件路径
- extract_dir：解压至的文件夹路径。文件夹可以不存在，会自动生成
- format：解压格式，默认为 None，会根据扩展名自动选择解压格式

```
import shutil,os
zip_path = os.path.join(os.getcwd(),"aaa.zip")
extract_dir = os.path.join(os.getcwd(),"test")
shutil.unpack_archive(zip_path, extract_dir)
```

get_unpack_formats()： 获取支持的解压文件格式。目前支持的有：tar、zip、gztar、bztar 和 xztar。Python3 新增方法

6. 数据库

6.1. Pycharm 连接 mysql 失败解决方案

关联 mysql 失败_Server returns invalid timezone. Go to 'Advanced' tab and set 'serverTimezon'
时区错误，MySQL 默认的时区是 UTC 时区，比北京时间晚 8 个小时。

所以要修改 mysql 的时长

在 mysql 的命令模式下，输入：

set global time_zone='+8:00';

再次连接成功

6.2. mysql-connector 驱动

mysql-connector 是 **MySQL** 官方提供的驱动器。

安装 **mysql-connector**：

```
pip install mysql-connector
```

```
import mysql.connector
```

执行以上代码，如果没有产生错误，表明安装成功。

6.2.1. 创建数据库连接

可以使用以下代码来连接数据库：

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",  # 数据库主机地址
    user="root",  # 数据库用户名
    passwd="123456"  # 数据库密码
)

print(mydb)
```

6.2.2. 创建数据库

创建数据库使用 "CREATE DATABASE" 语句，以下创建一个名为 test_db 的数据库：

```
mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE test_db")
```

创建数据库前我们也可以使用 "SHOW DATABASES" 语句来查看数据库是否存在：

输出所有数据库列表：

```
mycursor.execute("SHOW DATABASES")
print([x[0] for x in mycursor])
```

或者我们可以直接连接数据库，如果数据库不存在，会输出错误信息：

```
import mysql.connector

db = mysql.connector.connect(
    host="localhost",  # 数据库主机地址
    user="root",  # 数据库用户名
    passwd="123456",  # 数据库密码
    database="test_db"
)
```

6.2.3. 删除/创建数据表

删除表使用 **"DROP TABLE"** 语句， `IF EXISTS` 关键字是用于判断表是否存在，只有在存在的情况才删除：

```
cursor.execute("DROP TABLE IF EXISTS sites")
```

创建数据表使用 **"CREATE TABLE"** 语句，创建数据表前，需要确保数据库已存在，以下创建一个名为 EMPLOYEE 的数据表：

```
cursor.execute("CREATE TABLE sites (name VARCHAR(255), url VARCHAR(255))")
```

我们也可以使用 "SHOW TABLES" 语句来查看数据表是否已存在：

```
mycursor.execute("SHOW TABLES")
print([t[0] for t in mycursor])
```

6.2.4. 添加主键

创建表的时候我们一般都会设置一个主键（PRIMARY KEY），我们可以使用 **"INT AUTO_INCREMENT PRIMARY KEY"** 语句来创建一个主键，主键起始值为 1，逐步递增。

如果我们的表已经创建，我们需要使用 **ALTER TABLE** 来给表添加主键：

```
cursor.execute("ALTER TABLE sites ADD COLUMN id INT AUTO_INCREMENT PRIMARY KEY")
```

如果你还未创建 sites 表，可以直接使用以下代码创建：

```
cursor.execute("DROP TABLE IF EXISTS sites")
cursor.execute("CREATE TABLE sites (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), url
VARCHAR(255))")
```

6.2.5. 插入数据

插入数据使用 **"INSERT INTO"** 语句：

```
sql = "INSERT INTO sites (name, url) VALUES (%s, %s)"
val = ("Baidu", "https://www.baidu.com")
cursor.execute(sql, val)

db.commit()  # 数据表内容有更新，必须使用到该语句

print(cursor.rowcount, "记录插入成功。")
```

6.2.6. 批量插入

批量插入使用 **executemany()** 方法，该方法的第二个参数是一个元组列表，包含了我们要插入的数据：

向 sites 表插入多条记录。

```
cursor = db.cursor()
sql = "INSERT INTO sites (name, url) VALUES (%s, %s)"
val = [
    ('Google', 'https://www.google.com'),
    ('Github', 'https://www.github.com'),
    ('Taobao', 'https://www.taobao.com'),
    ('stackoverflow', 'https://www.stackoverflow.com/')
]

cursor.executemany(sql, val)


db.commit()  # 数据表内容有更新，必须使用到该语句



print(cursor.rowcount, "记录插入成功。")
```

执行代码，输出结果为：

```
4 记录插入成功。
```

执行以上代码后，我们可以看看数据表的记录：

id	name	url
1	Baidu	https://www.baidu.com
2	Google	https://www.google.com
3	Github	https://www.github.com
4	Taobao	https://www.taobao.com
5	stackoverflow	https://www.stackoverflow.com/
(Auto)	*(NULL)*	*(NULL)*

如果我们想在数据记录插入后，获取该记录的 ID ，可以使用以下代码：

```
sql = "INSERT INTO sites (name, url) VALUES (%s, %s)"
val = ("Zhihu", "https://www.zhihu.com")
cursor.execute(sql, val)

db.commit()

print("1 条记录已插入, ID:", cursor.lastrowid)
```

执行代码，输出结果为：

```
1 条记录已插入, ID: 6
```

6.2.7. 查询数据

- fetchone()：该方法获取下一个查询结果集。结果集是一个对象
- fetchall():接收全部的返回结果行.
- rowcount：这是一个只读属性，并返回执行 execute()方法后影响的行数。

查询数据使用 **SELECT** 语句：

```
cursor.execute("SELECT * FROM sites")


result = cursor.fetchall()  # fetchall() 获取所有记录



for x in result:
    print(x)
```

执行代码，输出结果为：

```
(1, 'Baidu', 'https://www.baidu.com')
(2, 'Google', 'https://www.google.com')
(3, 'Github', 'https://www.github.com')
(4, 'Taobao', 'https://www.taobao.com')
(5, 'stackoverflow', 'https://www.stackoverflow.com/')
(6, 'Zhihu', 'https://www.zhihu.com')
```

如果我们只想读取一条数据，可以使用 fetchone() 方法：

```
cursor.execute("SELECT * FROM sites")

myresult = cursor.fetchone()
print(myresult)
```

执行代码，输出结果为：

```
(1, 'Baidu', 'https://www.baidu.com')
```

为了防止数据库查询发生 SQL 注入的攻击，我们可以使用 `%s` 占位符来转义查询的条件：

```
sql = "SELECT * FROM sites WHERE name = %s"
na = ("Zhihu",)
cursor.execute(sql, na)
```

```
myresult = cursor.fetchall()

print([x for x in myresult])
```

6.2.8. Limit

如果我们要设置查询的数据量，可以通过 **"LIMIT"** 语句来指定

读取前 3 条记录：

```
mycursor.execute("SELECT * FROM sites LIMIT 3")
myresult = mycursor.fetchall()
for x in myresult:
    print(x)
```

也可以指定起始位置，使用的关键字是 **OFFSET**：

从第二条开始读取前 3 条记录：

```
mycursor.execute("SELECT * FROM sites LIMIT 3 OFFSET 1")  # 0 为 第一条，1 为第二条，以此类推

myresult = mycursor.fetchall()
for x in myresult:
    print(x)
```

6.2.9. 删除记录

删除 name 为 stackoverflow 的记录：

```
sql = "DELETE FROM sites WHERE name = 'stackoverflow'"
mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, " 条记录删除")
```

执行代码，输出结果为：

```
1  条记录删除
```

注意：要慎重使用删除语句，删除语句要确保指定了 WHERE 条件语句，否则会导致整表数据被删除。

为了防止数据库查询发生 SQL 注入的攻击，我们可以使用 `%s` 占位符来转义删除语句的条件：

```
mycursor.execute("DELETE FROM sites WHERE name = %s",  ("stackoverflow",))
mydb.commit()
print(mycursor.rowcount, " 条记录删除")
```

6.2.10. 更新表数据

UPDATE 语句要确保指定了 WHERE 条件语句，否则会导致整表数据被更新。

数据表更新使用 **"UPDATE"** 语句：

将 name 为 Zhihu 的字段数据改为 ZH：

```
sql = "UPDATE sites SET name = %s WHERE name = %s"
val = ("Zhihu", "ZH")
mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, " 条记录被修改")
```

6.3. PyMySQL 驱动

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2 中则使用 mysqldb。
PyMySQL 遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。

6.3.1. PyMySQL 安装

```
pip3 install PyMySQL
```

还可以使用 git 命令下载安装包安装(也可以手动下载)：
$ git clone https://github.com/PyMySQL/PyMySQL
$ cd PyMySQL/
$ python3 setup.py install

注意：请确保您有 root 权限来安装上述模块。
安装的过程中可能会出现"ImportError: No module named setuptools"的错误提示，意思是没有安装 setuptools，可以访问 https://pypi.python.org/pypi/setuptools 找到各个系统的安装方法。

6.3.2. 数据库连接

```
import pymysql

# 打开数据库连接

db = pymysql.connect("localhost", "root", "123456", "test_db")

# 使用 cursor() 方法创建一个游标对象 cursor

cursor = db.cursor()

# 使用 execute()  方法执行 SQL 查询

cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取单条数据.

data = cursor.fetchone()
print("Database version : %s " % data)

# 关闭数据库连接

db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.5.50
```

6.3.3. 创建数据库表

```
import pymysql


# 打开数据库连接

db = pymysql.connect("localhost", "root", "123456", "test_db")

# 使用 cursor() 方法创建一个游标对象 cursor

cursor = db.cursor()

# 使用 execute() 方法执行 SQL，如果表存在则删除

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 使用预处理语句创建表

sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接

db.close()
```

6.3.4. 数据库插入操作

```
import pymysql


# 打开数据库连接

db = pymysql.connect("localhost", "root", "123456", "test_db")

# 使用 cursor() 方法创建一个游标对象 cursor

cursor = db.cursor()

# SQL 插入语句

sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:

    # 执行sql 语句

    cursor.execute(sql)
```

```
    # 提交到数据库执行
    db.commit()
except:
    # 如果发生错误则回滚
    db.rollback()
# 关闭数据库连接
db.close()
```

以下方式防止 sql 注入:

```
import pymysql


# 打开数据库连接
db = pymysql.connect("localhost", "root", "123456", "test_db")
# 使用 cursor() 方法创建一个游标对象 cursor
cursor = db.cursor()
# SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES (%s, %s,  %s,  %s,  %s)"
val = ('Mac', 'Mohan', 20, 'M', 2000)

try:
    # 执行sql 语句
    cursor.execute(sql, val)
    # 执行sql 语句
    db.commit()
except Exception as e:
    print(e)
    # 发生错误时回滚
    db.rollback()


# 关闭数据库连接
db.close()
```

6.3.5. 数据库查询操作

Python 查询 Mysql 使用 fetchone() 方法获取单条数据，使用 fetchall() 方法获取多条数据。

- **fetchone():**该方法获取下一个查询结果集。结果集是一个对象
- **fetchall():**接收全部的返回结果行.

- **rowcount:**这是一个只读属性，并返回执行 execute()方法后影响的行数。

查询 EMPLOYEE 表中 salary（工资）字段大于 1000 的所有数据：

```
import pymysql


# 打开数据库连接

db = pymysql.connect("localhost", "root", "123456", "test_db")

# 使用 cursor() 方法创建一个游标对象 cursor

cursor = db.cursor()

# SQL 查询语句

sql = "SELECT * FROM EMPLOYEE WHERE INCOME > %s"
args = 1000
try:

    # 执行SQL 语句

    cursor.execute(sql, args)

    # 获取所有记录列表

    results = cursor.fetchall()
    for row in results:
        fname, lname, age, sex, income = row
        print("fname=%s,lname=%s,age=%s,sex=%s,income=%s" % \
              (fname, lname, age, sex, income))
except BaseException as e:
    print(e)


# 关闭数据库连接

db.close()
```

以上脚本执行结果如下：

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

6.3.6. 数据库更新操作

更新操作用于更新数据表的的数据，以下实例将 TESTDB 表中 SEX 为 'M' 的 AGE 字段递增 1：

```
import pymysql


# 打开数据库连接

db = pymysql.connect("localhost", "root", "123456", "test_db")

# 使用 cursor() 方法创建一个游标对象 cursor

cursor = db.cursor()
```

```
# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = %s"
args = 'M'
try:
    # 执行SQL 语句
    cursor.execute(sql,args)
    # 提交到数据库执行
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()

# 关闭数据库连接
db.close()
```

6.3.7. 删除操作

删除操作用于删除数据表中的数据，以下实例演示了删除数据表 EMPLOYEE 中 AGE 大于 20 的所有数据：

```
import pymysql

# 打开数据库连接
db = pymysql.connect("localhost", "root", "123456", "test_db")
# 使用 cursor() 方法创建一个游标对象 cursor
cursor = db.cursor()
# SQL 删除语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > %s"
args = 20
try:
    # 执行SQL 语句
    cursor.execute(sql,args)
    # 提交修改
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()
```

```
# 关闭连接
db.close()
```

6.3.8. 执行事务

事务机制可以确保数据一致性。

事务应该具有 4 个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为 ACID 特性。

- 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 commit 或 rollback：

```
# SQL 删除语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > %s"
args = 20
try:
   # 执行SQL 语句
   cursor.execute(sql,args)
   # 提交修改
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()
```

对于支持事务的数据库， 在 Python 数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。commit()方法游标的所有更新操作，rollback()方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

6.4. pyhton 操作数据库常见的错误及异常

DB API 中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常:

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。
Error	警告以外所有其他错误类。必须是 StandardError 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是 Error 的子

	类。
DatabaseError	和数据库有关的错误发生时触发。 必须是 Error 的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是 DatabaseError 的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、 数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。 必须是 DatabaseError 的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是 DatabaseError 子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是 DatabaseError 子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL 语句语法错误、 参数数量错误等等。必须是 DatabaseError 的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或 API 等。例如在连接对象上 使用.rollback() 函数，然而数据库并不支持事务或者事务已关闭。 必须是 DatabaseError 的子类。

6.5. SQLite

SQLite 是一种嵌入式数据库，它的数据库就是一个本地文件。
Python 就内置了 SQLite3，所以，在 Python 中使用 SQLite，不需要安装任何东西，直接使用。

```
import sqlite3


# 连接到SQLite 数据库,数据库文件是 test.db

# 如果文件不存在，会自动在当前目录创建:

conn = sqlite3.connect('test.db')

cursor = conn.cursor()
cursor.execute('create table user (id varchar(20) primary key, name varchar(20))')
cursor.execute('insert into user (id, name) values (\'1\', \'xiaohua\')')
print('rowcount =', cursor.rowcount)

cursor.close()
conn.commit()
conn.close()
```

查询记录：

```
import sqlite3

conn = sqlite3.connect('test.db')
cursor = conn.cursor()

# 执行查询语句:
```

```
cursor.execute('select * from user where id=?', '1')

# 获得查询结果集:

values = cursor.fetchall()
print(values)
cursor.close()
conn.close()
```

如果 SQL 语句带有参数，那么需要把参数按照位置传递给 execute()方法，有几个?占位符就必须对应几个参数，例如：

```
cursor.execute('select * from user where name=? and pwd=?', ('abc', 'password'))
```

SQLite 支持常见的标准 SQL 语句以及几种常见的数据类型。
MySQL 的 SQL 占位符是%s，而 SQLite 的 SQL 占位符是?

6.6. ORM 框架 SQLAlchemy

ORM 技术：Object-Relational Mapping，把关系数据库的表结构映射到对象上。
在 Python 中，最有名的 ORM 框架是 SQLAlchemy：

```
from sqlalchemy import Column, String, create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base


# 创建对象的基类:

Base = declarative_base()

# 定义 User 对象:

class User(Base):

    # 表的名字:

    __tablename__ = 'user'

    # 表的结构:

    id = Column(String(20), primary_key=True)
    name = Column(String(20))


# 初始化数据库连接:

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

# 创建 DBSession 类型:

DBSession = sessionmaker(bind=engine)
```

`create_engine()`用来初始化数据库连接。SQLAlchemy 用一个字符串表示连接信息：

```
'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'
```

由于有了 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：

```
# 创建 session 对象:
session = DBSession()
# 创建新 User 对象:
new_user = User(id='5', name='Bob')
# 添加到 session:
session.add(new_user)
# 提交即保存到数据库:
session.commit()
# 关闭 session:
session.close()
```

可见，关键是获取 session，然后把对象添加到 session，最后提交并关闭。`DBSession` 对象可视为当前数据库连接。

有了 ORM，可以直接从数据库表中查询 `User` 对象。SQLAlchemy 提供的查询接口如下：

```
# 创建 Session:
session = DBSession()
# 创建 Query 查询，filter 是 where 条件，最后调用 one() 返回唯一行，如果调用 all() 则返回所有行:
user = session.query(User).filter(User.id=='5').one()
# 打印类型和对象的 name 属性:
print('type:', type(user))
print('name:', user.name)
# 关闭 Session:
session.close()
```

如果一个 User 拥有多个 Book，就可以定义一对多关系如下：

```
class User(Base):
    __tablename__ = 'user'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
    # 一对多:
    books = relationship('Book')
class Book(Base):
    __tablename__ = 'book'

    id = Column(String(20), primary_key=True)
    name = Column(String(20))
```

```
    # “多”的一方的 book 表是通过外键关联到 user 表的:
    user_id = Column(String(20), ForeignKey('user.id'))
```

User,Book 表建表语句：

```
CREATE TABLE `user` (
   `id` VARCHAR(20) NOT NULL,
   `name` VARCHAR(20) DEFAULT NULL,
   PRIMARY KEY (`id`)
) ENGINE=INNODB DEFAULT CHARSET=utf8

CREATE TABLE `book` (
   `id` INT(20) NOT NULL AUTO_INCREMENT,
   `name` VARCHAR(20) DEFAULT NULL,
   `user_id` VARCHAR(20) DEFAULT NULL,
   PRIMARY KEY (`id`),
   KEY `user_id` (`user_id`),
   CONSTRAINT `book_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `user` (`id`)
) ENGINE=INNODB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8
```

当查询一个 User 对象时，该对象的 books 属性将返回一个包含若干个 Book 对象的 list：

```
# 创建 session 对象:
session = DBSession()
for i in range(5):
    # 创建新 User 对象:
    new_user = Book(name='book' + str(i), user_id="2")
    # 添加到 session:
    session.add(new_user)
# 提交即保存到数据库:
session.commit()

user = session.query(User).filter(User.id == '2').one()
print(user.id)
print(user.name)
for book in user.books:
    print(book.id, book.name, book.user_id)
# 关闭 session:
session.close()
```

7. Python 面向对象

- 类(Class): 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- 方法：类中定义的函数。
- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量用于处理类及其实例对象的相关的数据。
- 方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
- 局部变量：定义在方法中的变量，只作用于当前实例的类。
- 实例变量：在类的声明中，属性是用变量来表示的。这种变量就称为实例变量，是在类声明的内部但是在类的其他成员方法之外声明的。
- 继承：即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 Dog 类型的对象派生自 Animal 类，这是模拟"是一个（is-a）"关系（例图，Dog 是一个 Animal）。
- 实例化：创建一个类的实例，类的具体对象。
- 对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

和其它编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python 中的类提供了面向对象编程的所有基本功能：类的继承机制允许多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

7.1. 面向对象基础

7.1.1. 创建类

实例，类的帮助信息可以通过 ClassName.__doc__查看:

```
class MyClass:
    """一个简单的类实例"""
    i = 12345
    def f(self):
        return 'hello world'

# 实例化类
x = MyClass()

# 访问类的属性和方法
print("MyClass 类的属性 i 为：", x.i)
print("MyClass 类的方法 f 输出为：", x.f())
print(MyClass.__doc__)
```

结果：

```
MyClass 类的属性 i 为： 12345
MyClass 类的方法 f 输出为： hello world
一个简单的类实例
```

实例：

```
# coding=utf-8

class Employee:
   '所有员工的基类'
   empCount = 0
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
      print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name, ", Salary: ", self.salary)
```

- empCount 变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 Employee.empCount 访问。
- 第一个方法__init__()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法
- 类内部的方法必须有一个额外的第一个参数名称, 按照惯例它的名称是 self。self 代表类的实例，而不是类。self 不是 python 关键字，把它换成其他的变量名也是可以正常执行的。

实例：

```
class Demo:
   def prt(self):
      print(self)
      print(self.__class__)

t = Demo()
t.prt()
```

结果：

```
<__main__.Demo object at 0x0000000002113BA8>
<class '__main__.Demo'>
```

从运行结果可以看出：self 代表的是类的实例，代表当前对象的地址，而 self.class 则指向类。

7.1.2. 创建实例对象

实例化类其他编程语言中一般用关键字 new，但是在 Python 中并没有这个关键字，类的实例化类似函数调用方式。以下使用类的名称 Employee 来实例化，并通过 __init__ 方法接受参数。

```
"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
"创建 Employee 类的第二个对象"
```

```
emp2 = Employee("Manni", 5000)
```

7.1.3. 访问对象的属性

完整示例

```
# coding=utf-8

class Employee:
    '所有员工的基类'
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
    def displayCount(self):
        print("Total Employee %d" % Employee.empCount)
    def displayEmployee(self):
        print("Name : ", self.name, ", Salary: ", self.salary)

# "创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
# "创建 Employee 类的第二个对象"
emp2 = Employee("Manni", 5000)

emp1.displayEmployee()
emp2.displayEmployee()
print(emp1.name, emp1.salary)
print(emp2.name, emp2.salary)
print("Total Employee %d" % Employee.empCount)
```

运行结果：

```
Name :  Zara , Salary:  2000
Name :  Manni , Salary:  5000
Zara 2000
Manni 5000
Total Employee 2
```

添加，删除，修改类的属性

```
emp1.age = 7  # 添加一个 'age' 属性
emp1.age = 8  # 修改 'age' 属性
emp1.name="Jack" # 修改 'name' 属性
del emp1.name  # 删除 'name' 属性
del emp1.age  # 删除 'age' 属性
```

也可以使用以下函数的方式来访问属性：

- getattr(obj, name[, default])：访问对象的属性。
- hasattr(obj,name)：检查是否存在一个属性。
- setattr(obj,name,value)：设置一个属性。如果属性不存在，会创建一个新属性。

- delattr(obj, name)：删除属性。

```
hasattr(emp1, 'age')    # 如果存在 'age' 属性返回 True。
getattr(emp1, 'age')    # 返回 'age' 属性的值
setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8
delattr(emp1, 'age')    # 删除属性 'age'
```

7.1.4. Python 内置类属性

- __dict__：类的属性（包含一个字典，由类的数据属性组成）
- __doc__ :类的文档字符串
- __name__：类名
- __module__：类定义所在的模块（类的全名是'__main__.className'，如果类位于一个导入模块 mymod 中，那么 className.__module__ 等于 mymod）
- __bases__：类的所有父类构成元素（包含了一个由所有父类组成的元组）

实例：

```
# coding=utf-8

class Employee:
    '所有员工的基类'
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
    def displayCount(self):
        print("Total Employee %d" % Employee.empCount)
    def displayEmployee(self):
        print("Name : ", self.name, ", Salary: ", self.salary)

print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

运行结果：

```
Employee.__doc__: 所有员工的基类
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', '__doc__': '所有员工的基类', 'empCount': 0, '__init__': <function Employee.__init__ at 0x0000000002526488>, 'displayCount': <function Employee.displayCount at 0x000000000355EAE8>, 'displayEmployee': <function Employee.displayEmployee at 0x000000000355EB70>, '__dict__': <attribute '__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>}
```

7.1.5. python 对象销毁(垃圾回收)

Python 使用了引用计数这一简单技术来跟踪和回收垃圾。
在 Python 内部记录着所有使用中的对象各有多少引用。
一个内部跟踪变量，称为一个引用计数器。
当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也就是说， 这个对象的引用计数变为 0 时， 它被垃圾回收。但是回收不是"立即"的， 由解释器在适当的时机，将垃圾对象占用的内存空间回收。

```
a = 40      # 创建对象 〈40〉
b = a       # 增加引用， 〈40〉 的计数
c = [b]     # 增加引用. 〈40〉 的计数

del a       # 减少引用 〈40〉 的计数
b = 100     # 减少引用 〈40〉 的计数
c[0] = -1   # 减少引用 〈40〉 的计数
```

垃圾回收机制不仅针对引用计数为 0 的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充， 垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。 在这种情况下， 解释器会暂停下来， 试图清理所有未引用的循环。

析构函数 __del__ ，__del__在对象销毁的时候被调用，当对象不再被使用时，__del__方法运行：
实例：

```
# coding=utf-8

class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __del__(self):
        class_name = self.__class__.__name__
        print("实例%s 被销毁,所属类是%s" % (self, class_name))

pt1 = Point()  # 创建一个对象
pt3 = pt2 = pt1
print(id(pt1), id(pt2), id(pt3))  # 打印对象的 id
del pt1
del pt2
del pt3
```

结果：

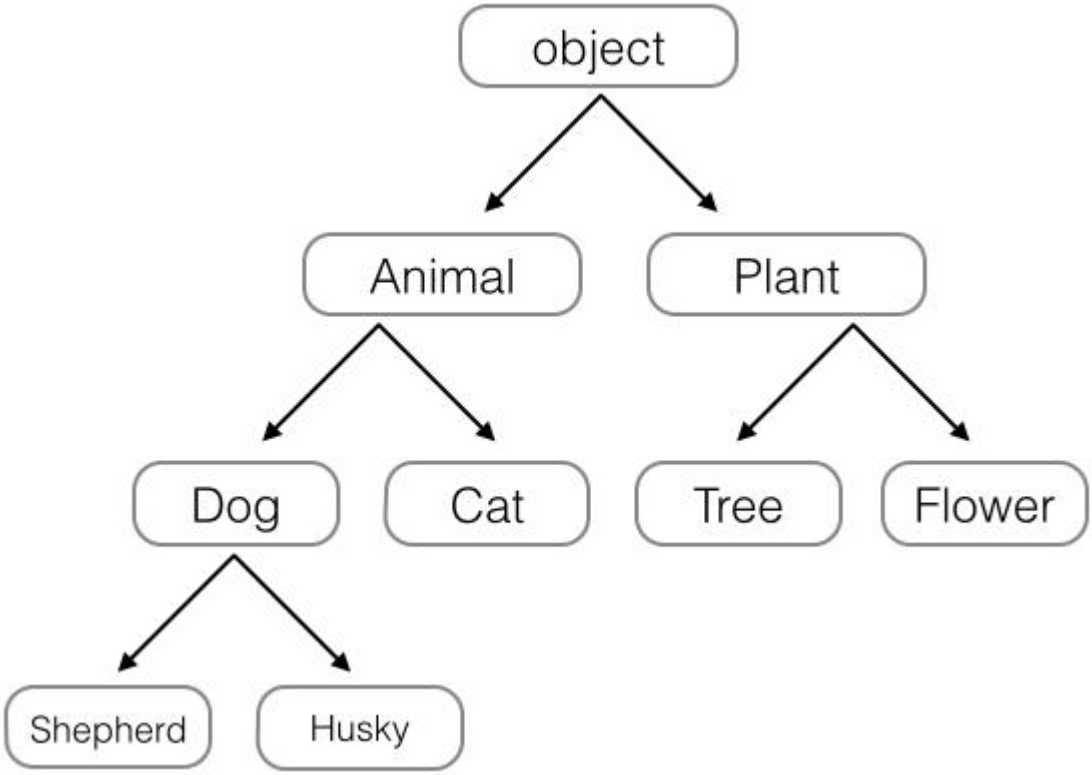
```
39234920 39234920 39234920
实例<__main__.Point object at 0x000000000256AD68>被销毁,所属类是 Point
```

上例中，先创建了一个 Point 对象,然后将 pt1、pt2、pt3 的引用均指向该对象
打印 id，发现确实都是同一个对象。
删除这三个引用时，就是断开对这个对象的引用
当三个引用全部被断开时，这个对象已经没有被任何变量引用变成"垃圾"

从而可能触发垃圾回收，在垃圾回收的时候会运行对象的__del__方法

7.1.6. 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。
继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类 object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写；
有了继承，才能有多态。在调用类实例方法的时候，尽量把变量视作父类类型，这样，所有子类类型都可以正常被接收。

语法：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

在 python 中继承中的一些特点：

- 1：在继承中父类的构造（__init__()方法）不会被自动调用，它需要在其子类的构造中亲自专门调用。
- 2：在调用父类的方法时，需要加上父类的类名前缀，且需要带上 self 参数变量。区别于在类中调用普通函数时并不需要带上 self 参数
- 3：Python 总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。
- 4：如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" 。

实例：

```
# coding=utf-8
```

```
class Parent:  # 定义父类
    parentAttr = 100

    def __init__(self):
        print("调用父类构造函数")

    def parentMethod(self):
        print('调用父类方法')

    def method(self):
        print('调用父类 method 方法')


class Child(Parent):  # 定义子类
    def __init__(self):
        super().__init__()
        print("调用子类构造方法")

    def childMethod(self):
        print('调用子类方法 child method')

    def method(self):
        super().method()
        print('调用子类 method 方法')


c = Child()  # 实例化子类
c.childMethod()  # 调用子类的方法
c.parentMethod()  # 调用父类方法
c.method()  # 调用覆写的方法
```

结果：

```
调用父类构造函数
调用子类构造方法
调用子类方法 child method
调用父类方法
调用父类 method 方法
调用子类 method 方法
```

7.1.7. 多重继承

```
class A:  # 定义类 A
    pass

class B:  # 定义类 B
    pass

class C(A, B):  # 继承类 A 和 B
```

```
    pass
```

检测函数：

- issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)
- isinstance(obj, Class) 布尔函数如果 obj 是 Class 类的实例对象或者是一个 Class 子类的实例对象则返回 true。

```
print(issubclass(C,A))
print(issubclass(C,B))
print(issubclass(A,B))
print(issubclass(A,C))
```

```
True
True
False
False
```

```
objA, objB, objC = A(), B(), C()
print(isinstance(objA, C),isinstance(objA, B))
print(isinstance(objB, A),isinstance(objB, C))
print(isinstance(objC, A),isinstance(objC, B))
```

```
False False
False False
True True
```

示例：

```
# 类定义
class people:
    # 定义基本属性
    name = ''
    age = 0
    # 定义私有属性,私有属性在类外部无法直接进行访问
    __weight = 0

    # 定义构造方法
    def __init__(self, n, a, w):
        self.name = n
        self.age = a
        self.__weight = w

    def speak(self):
        print("%s 说: 我 %d 岁。" % (self.name, self.age))

# 单继承示例
class student(people):
    grade = ''

    def __init__(self, n, a, w, g):
        # 调用父类的构函
        people.__init__(self, n, a, w)
        self.grade = g

    # 覆写父类的方法
```

```
    def speak(self):
        print("%s 说: 我 %d 岁了，我在读 %d 年级" % (self.name, self.age, self.grade))


# 另一个类，多重继承之前的准备
class speaker():
    topic = ''
    name = ''

    def __init__(self, n, t):
        self.name = n
        self.topic = t

    def speak(self):
        print("我叫 %s，我是一个演说家，我演讲的主题是 %s" % (self.name, self.topic))


# 多重继承
class sample(speaker, student):
    a = ''

    def __init__(self, n, a, w, g, t):
        student.__init__(self, n, a, w, g)
        speaker.__init__(self, n, t)


test = sample("Tim", 25, 80, 4, "Python")
test.speak()  # 方法名同，默认调用的是在括号中排前地父类的方法
```

如果父类 A 和父类 B 中，有一个同名的方法，那么通过子类去调用的时候，调用哪个？

```
class base(object):
    def test(self):
        print('----base test----')

class A(base):
    def test(self):
        print('----A test----')


# 定义一个父类

class B(base):
    def test(self):
        print('----B test----')


# 定义一个子类，继承自A、B

class C(A,B):
    pass
```

```
obj_C = C()
obj_C.test()


print(C.__mro__) #可以查看 C 类的对象搜索方法时的先后顺序
```

结果：

```
----A test----
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class
'__main__.base'>, <type 'object'>)
```

7.1.8. 方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

```
class Parent:  # 定义父类
    def myMethod(self):
        print('调用父类方法')


class Child(Parent):  # 定义子类
    def myMethod(self):
        print('调用子类方法')


c = Child()  # 子类实例
c.myMethod()  # 子类调用重写方法
super(Child, c).myMethod()  # 用子类对象调用父类已被覆盖的方法
```

```
调用子类方法
调用父类方法
```

7.1.9. 类属性与方法

类的私有属性

__private_attrs：两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。在类内部的方法中使用时 self.__private_attrs。

类的方法

在类地内部，使用 def 关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数 self,且为第一个参数

类的私有方法

__private_method：两个下划线开头，声明该方法为私有方法，不能在类地外部调用。在类的内部调用 self.__private_methods

```
# coding=utf-8

class JustCounter:
    __secretCount = 0  # 私有变量
```

```
    publicCount = 0  # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print(self.__secretCount)


counter = JustCounter()
counter.count()
counter.count()
print(counter.publicCount)
print(counter.__secretCount)  # 报错，实例不能访问私有变量
```

运行结果：

```
1
2
2
Traceback (most recent call last):
  File "D:/PycharmProjects/demo1/pythonlearn/test.py", line 17, in <module>
    print(counter.__secretCount)  # 报错，实例不能访问私有变量
AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Python 不允许实例化的类访问私有数据，但你可以使用 **object._className__attrName** 访问属性，但是强烈建议不要这么干，因为不同版本的 Python 解释器可能会把**__attrName** 改成不同的变量名。

将如下代码替换以上代码的最后一行代码：

```
print(counter._JustCounter__secretCount)
```

再运行，结果：

```
1
2
2
2
```

__foo__: 定义的是内置方法，类似 __init__() 之类的。

_foo: 以单下划线开头的表示的是 protected 类型的变量

即保护类型只能允许其本身与子类进行访问，不能用于 from module import *

__foo: 双下划线的表示的是私有类型(private)的变量, 只能是允许这个类本身进行访问了。

7.1.10. getattr、setattr 及 hasattr

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

```
class MyObject(object):
    def __init__(self):
        self.x = 9
    def power(self):
        return self.x * self.x
```

```
obj = MyObject()
```

紧接着，可以测试该对象的属性：

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

如果试图获取不存在的属性，会抛出 AttributeError 的错误：

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

可以传入一个 default 参数，如果属性不存在，就返回默认值：

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

也可以获得对象的方法：

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

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

```
sum = obj.x + obj.y
```

就不要写：

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

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
```

```
    return None
```

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

7.2. 静态方法和类方法

7.2.1. 类方法

是类对象所拥有的方法，需要用修饰器@classmethod 来标识其为类方法，对于类方法，第一个参数必须是类对象，一般以 cls 作为第一个参数（当然可以用其他名称的变量作为其第一个参数，但是大部分人都习惯以'cls'作为第一个参数的名字，就最好用'cls'了），能够通过实例对象和类对象去访问。

```
class People(object):
    country = 'china'


    #类方法，用 classmethod 来进行修饰
    @classmethod
    def getCountry(cls):
        return cls.country

p = People()
print(p.getCountry())    #可以用过实例对象引用

print(People.getCountry())    #可以通过类对象引用
```

结果：
china
china

类方法还有一个用途就是可以对类属性进行修改：

```
class People:
    country = 'china'


    #类方法，用 classmethod 来进行修饰
    @classmethod
    def getCountry(cls):
        return cls.country

    @classmethod
    def setCountry(cls,country):
        cls.country = country
```

```
p = People()

print(p.getCountry())    #可以用过实例对象引用

print(People.getCountry())    #可以通过类对象引用



p.setCountry('japan')

print(p.getCountry())
print(People.getCountry())
```

结果：
china
china
japan
japan

结果显示在用类方法对类属性修改之后，通过类对象和实例对象访问都发生了改变

7.2.2. 静态方法

需要通过修饰器@staticmethod 来进行修饰，静态方法不需要多定义参数

```
class People:
    country = 'china'


    # 静态方法

    @staticmethod
    def getCountry():
        return People.country

print(People.getCountry())
```

7.2.3. 总结

从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象 cls，那么通过 cls 引用的必定是类对象的属性和方法；而实例方法的第一个参数是实例对象 self，那么通过 self 引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用

7.3. 动态绑定与限制

7.3.1. 动态绑定属性

```
class Student(object):
    pass

s = Student()
s.name = 'Michael'  # 动态给实例绑定一个属性



def set_age(self, age):  # 定义一个函数作为实例方法
    self.age = age

from types import MethodType


s.set_age = MethodType(set_age, s)  # 给实例绑定一个方法

s.set_age(25)  # 调用实例方法

print(s.age)  # 测试结果
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
s2 = Student() # 创建新的实例
s2.set_age(25)
```

```
Traceback (most recent call last):
  File "D:/PycharmProjects/demo1/pythonlearn/tmp.py", line 17, in <module>
    s2.set_age(25)
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给 class 绑定方法：

```
def set_score(self, score):
    self.score = score
Student.set_score = MethodType(set_score, Student)
```

给 class 绑定方法后，所有实例均可调用：

```
s.set_score(100)
print(s.score)
s2 = Student()
s2.set_score(99)
print(s2.score)
```

通常情况下，上面的 `set_score` 方法可以直接定义在 class 中，但动态绑定允许我们在程序运行的过程中动态给 class 加上功能，这在静态语言中很难实现。

7.3.2. __slots__ 限制绑定

如果我们想要限制 class 的属性，比如只允许对 Student 实例添加 name 和 age 属性。
为了达到限制的目的，Python 允许在定义 class 的时候，定义一个特殊的__slots__变量，来限制该 class 能添加的属性：

```
class Student(object):
    __slots__ = ('name', 'age')
```

然后，我们试试：

```
s = Student() # 创建新的实例

s.name = 'xiaoming' # 绑定属性'name'

s.age = 25 # 绑定属性'age'

s.score = 99 # 绑定属性'score'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于'score'没有被放到__slots__中，所以不能绑定 score 属性，试图绑定 score 将得到 AttributeError 的错误。
使用__slots__要注意，__slots__定义的属性仅对当前类起作用，对继承的子类是不起作用的：
除非在子类中也定义__slots__，这样，子类允许定义的属性就是自身的__slots__加上父类的__slots__。也就是说没有定义__slots__时，__slots__的值是全部。

7.4. 使用@property 将方法变成属性调用

语法:

```
class property([fget[, fset[, fdel[, doc]]]])
```

参数：

- fget -- 获取属性值的函数
- fset -- 设置属性值的函数
- fdel -- 删除属性值函数
- doc -- 属性描述信息

返回新式类属性。
将 property 函数用作装饰器可以很方便的创建只读属性：

```
class Parrot(object):
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

上面的代码将 voltage() 方法转化成同名只读属性的 getter 方法。
property 的 getter,setter 和 deleter 方法同样可以用作装饰器：

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

7.5. 类的专有方法

__init__ : 构造函数，在生成对象时调用
__del__ : 析构函数，释放对象时使用
__repr__ : 打印，转换
__str__(self) 用于将值转化为适于人阅读的形式
__setitem__ : 按照索引赋值
__getitem__: 按照索引获取值
__len__: 获得长度
__call__: 函数调用
__add__: 加运算
__sub__: 减运算
__mul__: 乘运算
__truediv__: 除运算
__mod__: 求余运算
__pow__: 乘方
基础重写方法：

基本示例：

```
import math

class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%s, %s)' % (self.a, self.b)

    def __repr__(self):
        return self.__str__()
```

```
    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)
    def __sub__(self, other):
        return Vector(self.a - other.a, self.b - other.b)
    def __mul__(self, other):
        return Vector(self.a * other.a, self.b * other.b)
    def __truediv__(self, other):
        return Vector(self.a / other.a, self.b / other.b)
    def __mod__(self, other):
        return Vector(self.a % other.a, self.b % other.b)
    def __pow__(self, other):
        return Vector(self.a ** other.a, self.b ** other.b)

v1 = Vector(2, 10)
v2 = Vector(5, -2)
print(v1 + v2)
print(v1 - v2)
print(v1 * v2)
print(v1 / v2)
print(v1 % v2)
print(v1 ** v2)
```

结果：

Vector (7, 8)
Vector (-3, 12)
Vector (10, -20)
Vector (0.4, -5.0)
Vector (2, 0)
Vector (32, 0.01)

```
sorted_key = lambda v: math.sqrt(v.a ** 2 + v.b ** 2)
l = sorted([Vector(2, 10), Vector(3.5, 10), Vector(5, 8), Vector(4, 3)], key=sorted_key)
print(l)
```

结果：

[Vector (4, 3), Vector (5, 8), Vector (2, 10), Vector (3.5, 10)]

7.5.1. __iter__

如果一个类想被用于循环 `for...in`，类似 list 或 tuple 那样，就必须实现一个方法`__iter__()`，该方法返回一个迭代对象，然后，Python 的 for 循环就会不断调用该迭代对象的`next()`方法拿到循环的下一个值，直到遇到 StopIteration 错误时退出循环。

我们以斐波那契数列为例，写一个 Fib 类，可以作用于 for 循环：

```
class Fib(object):
    def __init__(self):

        self.a, self.b = 0, 1  # 初始化两个计数器 a，b

    def __iter__(self):
```

```
        return self  # 实例本身就是迭代对象，故返回自己
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b  # 计算下一个值
        if self.a > 100000:  # 退出循环的条件
            raise StopIteration()
        return self.a  # 返回下一个值


for n in Fib():
    print(n, end=",")
```

1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,

7.5.2. __getitem__

要表现得像 list 那样按照下标取出元素，需要实现方法：`__getitem__()`

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了。
对于切片方法[5:10]，将传入 slice 对象，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int):
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice):
            start = n.start
            stop = n.stop
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

这个例子没有对 step 参数和负数作处理，所以，要正确实现一个__getitem__()还是有很多工作要做的。

7.5.3. __getattr__

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。

要避免这个错误，除了可以加上一个相应属性外，Python 还有另一个机制，那就是写一个__getattr__()方法，动态返回一个属性。修改如下：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如 score，Python 解释器会试图调用__getattr__(self, 'score')来尝试获得属性。
返回函数也完全可以：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
```

注意，只有在没有找到属性的情况下，才调用__getattr__，已有的属性不会在__getattr__中查找。

链式调用示例：

```
# coding=utf-8

class Chain(object):
    def __init__(self, path=''):
        self._path = path

    def __getattr__(self, path):
        if path == 'users':
            return lambda user: Chain('%s/users/%s' % (self._path, user))
        return Chain('%s/%s' % (self._path, path))

    def __str__(self):
        return self._path

print(Chain().status.user.timeline.list)
print(Chain().status.users('xiaoxiaoming').repos)
```

结果：

```
/status/user/timeline/list
/users/xiaoxiaoming/repos
```

7.5.4. __call__实例方法

```
class Student(object):
    def __init__(self, name):
        self.name = name

    def __call__(self):
```

```
        print('My name is %s.' % self.name)
s = Student('Michael')
s()
```

__call__()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

通过 callable()函数可以判断一个对象是否是“可调用”对象：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('string')
False
```

7.5.5. __new__和__init__方法

__new__和__init__的作用

__new__至少要有一个参数 cls，代表要实例化的类，此参数在实例化时由 Python 解释器自动提供。**__new__必须要有返回值，返回实例化出来的实例。**

__init__有一个参数 self，就是这个__new__返回的实例，__init__在__new__的基础上可以完成一些其它初始化的动作，__init__不需要返回值。

代码：

```
# coding=utf-8

class A(object):
    def __init__(self):
        print("__init__内部：", self, id(self))

    def __new__(cls):
        print("__new__内部:", cls, id(cls))
        obj = object.__new__(cls)
        print("__new__方法返回值：", obj, id(obj))
        return obj


a = A()
print("主程序：", a, id(a))
print("-----------")
b = A()
print("主程序：", b, id(b))
```

运行结果：

```
__new__内部:<class '__main__.A'> 49546168
__new__方法返回值： <__main__.A object at 0x000000000225ABE0> 36023264
__init__内部： <__main__.A object at 0x000000000225ABE0> 36023264
主程序： <__main__.A object at 0x000000000225ABE0> 36023264
----------
__new__内部:<class '__main__.A'> 49546168
__new__方法返回值： <__main__.A object at 0x000000000225AD68> 36023656
__init__内部： <__main__.A object at 0x000000000225AD68> 36023656
主程序： <__main__.A object at 0x000000000225AD68> 36023656
```

可以看出，每次创建对象，都会先调用__new__方法，构建出对象，然后调用__init__方法，并将刚才创建的对象传入。

7.6. 单例模式

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，单例模式是一种对象创建型模式。

```python
# coding=utf-8


# 实例化一个单例

class Singleton(object):
    __instance = None

    def __new__(cls, age, name):

        # 如果类数字能够__instance 没有或者没有赋值

        # 那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个方法时

        # 能够知道之前已经创建过对象了，这样就保证了只有 1 个对象

        if not cls.__instance:
            cls.__instance = object.__new__(cls)
        return cls.__instance

a = Singleton(18, "dongGe")
b = Singleton(8, "dongGe")

print(id(a), id(b))


a.age = 19  # 给 a 指向的对象添加一个属性

print(b.age)  # 获取 b 指向的对象的 age 属性
```

结果：

```
38907296 38907296
19
```

创建单例时，只执行 1 次__init__方法：

```
# coding=utf-8

class Singleton(object):
    __instance = None
    __first_init = False

    def __new__(cls, age, name):
        if not cls.__instance:
            cls.__instance = object.__new__(cls)
        return cls.__instance

    def __init__(self, age, name):
        if not self.__first_init:
            self.age = age
            self.name = name
            Singleton.__first_init = True

    def __str__(self):
        return "Singleton[%s,%s]" % (self.name, self.age)


a = Singleton(18, "dongGe")
b = Singleton(8, "dongGe")

print(id(a), id(b))
print(a.age, b.age)
print(a, b)
```

运行结果：

```
39169496 39169496
18 18
Singleton[dongGe,18] Singleton[dongGe,18]
```

7.7. Enum 枚举类

基本实现：

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

这样我们就获得了 `Month` 类型的枚举类，可以直接使用 `Month.Jan` 来引用一个常量，或者枚举它的所有成员：

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

`value` 属性则是自动赋给成员的 `int` 常量，默认从 `1` 开始计数。

如果需要更精确地控制枚举类型，可以从 `Enum` 派生出自定义类：

```
@unique
class Weekday(Enum):
    Sun = 0
```

```
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6



for name, member in Weekday.__members__.items():
    print(name, '=>', member, ',', member.value)
```

`@unique`装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

```
Weekday = Enum('Weekday', ("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"))
for i in range(1,8):
    print(i,Weekday(i),Weekday(i).name)
print(Weekday.Tue)
print(Weekday['Tue'])
print(Weekday.Tue.value)
print(Weekday(1))
```

可见，既可以用成员名称引用枚举常量，又可以直接根据 value 的值获得枚举常量。

交通灯示例：

```
@unique
class TrafficLamp(Enum):
    RED = 60
    GREEN = 45
    YELLOW = 5

    def next(self):
        if self.name == 'RED':
            return TrafficLamp.GREEN
        elif self.name == 'GREEN':
            return TrafficLamp.YELLOW
        elif self.name == 'YELLOW':
            return TrafficLamp.RED

current = TrafficLamp.RED
for i in range(10):
    print(current, current.value)
    current = current.next()
```

7.8. 元类

7.8.1. type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个 Hello 的 class，就写一个 hello.py 模块：

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)
```

当 Python 解释器载入 hello 模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个 Hello 的 class 对象，测试如下：

type()函数可以查看一个类型或变量的类型，Hello 是一个 class，它的类型就是 type，而 h 是一个实例，它的类型就是 class Hello。

class 的定义是运行时动态创建的，而创建 class 的方法就是使用 type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过 type()函数创建出 Hello 类，而无需通过 class Hello(object)...的定义：

```
# 创建Hello class
Hello = type('Hello', (object,),
             dict(hello=lambda self, name='world': print('Hello, %s.' % name)))
h = Hello()
h.hello()
print(type(Hello))
print(type(h))
```

结果：

Hello, world.

<class 'type'>

<class '__main__.Hello'>

要创建一个 class 对象，type()函数依次传入 3 个参数：

- class 的名称；
- 继承的父类集合，注意 Python 支持多重继承，如果只有一个父类，别忘了 tuple 的单元素写法；
- class 的方法名称与函数绑定，这里我们把函数 fn 绑定到方法名 hello 上。

通过 `type()` 函数创建的类和直接写 class 是完全一样的，因为 Python 解释器遇到 class 定义时，仅仅是扫描一下 class 定义的语法，然后调用 `type()` 函数创建出 class。

正常情况下，我们都用 `class Xxx...` 来定义类，但是，`type()` 函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

7.8.2. __metaclass__属性

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据 metaclass 创建出类，所以：先定义 metaclass，然后创建类。

连接起来就是：先定义 metaclass，就可以创建类，最后创建实例。

所以，metaclass 允许你创建类或者修改类。换句话说，你可以把类看成是 metaclass 创建出来的“实例”。

```
class Foo(Bar):
    pass
```

对于上述代码 Python 做了如下的操作：

- Foo 中有__metaclass__这个属性吗？如果是，Python 会通过__metaclass__创建一个名字为 Foo 的类(对象)
- 如果 Python 没有找到__metaclass__，它会继续在 Bar（父类）中寻找__metaclass__属性，并尝试做和前面同样的操作。
- 如果 Python 在任何父类中都找不到__metaclass__，它就会在模块层次中去寻找__metaclass__，并尝试做同样的操作。
- 如果还是找不到__metaclass__,Python 就会用内置的 type 来创建这个类对象。

假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定__metaclass__：

```
def upper_attr(future_class_name, future_class_parents, future_class_attr):

    #遍历属性字典，把不是__开头的属性名字变为大写
    newAttr = {}
    for name,value in future_class_attr.items():
        if not name.startswith("__"):
            newAttr[name.upper()] = value

    #调用type来创建一个类
    return type(future_class_name, future_class_parents, newAttr)

class Foo(object, metaclass = upper_attr):
    bar = 'bip'

print(hasattr(Foo, 'bar'))
# 输出: False
print(hasattr(Foo, 'BAR'))
# 输出:True

f = Foo()
print(f.BAR)
# 输出:'bip'
```

```
False
True
bip
```

这一次改用一个真正的 class 来当做元类。

```
class UpperAttrMetaClass(type):
    # __new__ 是在__init__之前被调用的特殊方法
    # __new__是用来创建对象并返回之的方法
    # 而__init__只是用来将传入的参数初始化给对象
    # 你很少用到__new__，除非你希望能够控制对象的创建
    # 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写__new__
    # 如果你希望的话，你也可以在__init__中做些事情
    # 还有一些高级的用法会涉及到改写__call__特殊方法，但是我们这里不用
    def __new__(cls, future_class_name, future_class_parents, future_class_attr):
        #遍历属性字典，把不是__开头的属性名字变为大写
        newAttr = {}
        for name,value in future_class_attr.items():
            if not name.startswith("__"):
                newAttr[name.upper()] = value
```

```
        # 方法1：通过'type'来做类对象的创建
        # return type(future_class_name, future_class_parents, newAttr)

        # 方法2：复用type.__new__方法，这就是基本的OOP编程
        # return type.__new__(cls, future_class_name, future_class_parents,
newAttr)

        # 方法3：使用super方法
        return super(UpperAttrMetaClass, cls).__new__(cls, future_class_name,
future_class_parents, newAttr)


class Foo(object, metaclass = UpperAttrMetaClass):
    bar = 'bip'

print(hasattr(Foo, 'bar'))
# 输出: False
print(hasattr(Foo, 'BAR'))
# 输出:True

f = Foo()
print(f.BAR)
# 输出:'bip'
```

```
False
True
bip
```

就元类本身而言，它们其实是很简单的：

- 拦截类的创建
- 修改类
- 返回修改之后的类

7.8.3. 实现简易 ORM 框架

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个 ORM 框架，想定义一个 `User` 类来操作对应的数据库表 `User`，我们期待他写出这样的代码：

```
class User(Model):
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

u = User(id=12345, name='xiaoxiaoming', email='test@orm.org', password='my-pwd')
u.save()
```

其中，父类 `Model` 和属性类型 `StringField`、`IntegerField` 是由 ORM 框架提供的，剩下的魔术方法比如 `save()` 全部由 metaclass 自动完成。虽然 metaclass 的编写会比较复杂，但 ORM 的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该 ORM。

首先来定义 `Field` 类，它负责保存数据库表的字段名和字段类型：

```
class Field(object):

    def __init__(self, name, column_type):
        self.name = name
        self.column_type = column_type

    def __str__(self):
        return '<%s:%s>' % (self.__class__.__name__, self.name)
```

在 `Field` 的基础上，进一步定义各种类型的 `Field`，比如 `StringField`，`IntegerField` 等等：

```
class StringField(Field):

    def __init__(self, name):
        super(StringField, self).__init__(name, 'varchar(100)')

class IntegerField(Field):

    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')
```

下一步，编写 `ModelMetaclass`：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        print('Found model: %s' % name)
        mappings = {}
        for k, v in attrs.items():
            if isinstance(v, Field):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
        for k in mappings.keys():
            attrs.pop(k)
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        attrs['__table__'] = name # 假设表名和类名一致
        return type.__new__(cls, name, bases, attrs)
```

以及基类 `Model`：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)
```

```
    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v.name)
            params.append('%s')
            args.append(getattr(self, k))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join(params))
        print('SQL: %s , ARGS: %s' % (sql, str(args)))
```

当用户定义一个 `class User(Model)`时，Python 解释器首先在当前类 `User` 的定义中查找 `metaclass`，如果没有找到，就继续在父类 `Model` 中查找 `metaclass`，找到了，就使用 `Model` 中定义的 `metaclass` 的 `ModelMetaclass` 来创建 `User` 类，也就是说，metaclass 可以隐式地继承到子类，但子类自己却感觉不到。

在 `ModelMetaclass` 中，一共做了几件事情：

- 排除掉对 `Model` 类的修改；
- 在当前类（比如 `User`）中查找定义的类的所有属性，如果找到一个 Field 属性，就把它保存到一个`__mappings__` 的 dict 中，同时从类属性中删除该 Field 属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；
- 把表名保存到`__table__`中，这里简化为表名默认为类名。

在 Model 类中，就可以定义各种操作数据库的方法，比如 save()，delete()，find()，update 等等。
我们实现了 save()方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出 INSERT 语句。

```
u = User(id=12345, name='xiaoxiaoming', email='test@orm.org', password='my-pwd')
u.save()
```

输出如下：

```
Found model: User
Found mapping: id ==> <IntegerField:id>
Found mapping: name ==> <StringField:username>
Found mapping: email ==> <StringField:email>
Found mapping: password ==> <StringField:password>
SQL: insert into User (id,username,email,password) values (%s,%s,%s,%s) , ARGS: [12345, 'xiaoxiaoming', 'test@orm.org', 'my-pwd']
```

8. Python 常用库

8.1. 单元测试

为了编写单元测试，我们需要引入 Python 自带的 unittest 模块：

```
# -*- coding: utf-8 -*-

import unittest

class Dict(dict):

    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEqual(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEqual(d['key'], 'value')
```

```
    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty

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

编写单元测试时，我们需要编写一个测试类，从 `unittest.TestCase` 继承。

以 `test` 开头的方法就是测试方法，不以 `test` 开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个 `test_xxx()` 方法。由于 `unittest.TestCase` 提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 `assertEquals()`：

```
self.assertEquals(abs(-1), 1)  # 断言函数返回的结果与 1 相等
```

另一种重要的断言就是期待抛出指定类型的 Error，比如通过 `d['empty']` 访问不存在的 key 时，断言会抛出 `KeyError`：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过 d.empty 访问不存在的 key 时，我们期待抛出 AttributeError：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

8.1.1. 运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在 mydict_test.py 的最后加上两行代码：

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

这样就可以把 `mydict_test.py` 当做正常的 python 脚本运行：

```
$ python mydict_test.py
```

另一种更常见的方法是在命令行通过参数-m unittest 直接运行单元测试：

```
$ python -m unittest mydict_test
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

8.1.2. setUp 与 tearDown

可以在单元测试中编写两个特殊的 `setUp()`和 `tearDown()`方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()`和 `tearDown()`方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在 `setUp()`方法中连接数据库，在 `tearDown()`方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。
单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。
单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有 bug。
单元测试通过了并不意味着程序就没有 bug 了，但是不通过程序肯定有 bug。

8.2. 数据压缩

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

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

8.3. 性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多,timeit 证明了现代的方法更快一些。

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

```
# 0.07005939099999997
print(Timer('a,b = b,a', 'a=1; b=2').timeit())
# 0.027870997000000064
```

相对于 timeit 的细粒度，:mod:profile 和 pstats 模块提供了针对更大代码块的时间度量工具。

8.4. http 文件服务器

python -m http.server PORT

8.5. SMTP 发送邮件

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

python 的 smtplib 提供了一种很方便的途径发送电子邮件。它对 smtp 协议进行了简单的封装。

完整示例：

```
import smtplib
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.header import Header


# 第三方 SMTP 服务

from email.utils import formataddr


mail_host = "smtp.qq.com"       # 设置服务器

mail_user = "604049322@qq.com"  # 用户名

mail_pass = "aaaaaa"  # 口令



receiver = '15074804724@163.com'  # 接收邮箱



mail_msg = """
<p>Python 邮件发送测试...</p>
<p><a href="http://blog.xiaoxiaoming.xyz">网站链接</a></p>
<p>图片演示：</p>
<p><img src="cid:image1"></p>
"""
```

```
# 创建一个带附件的实例
message = MIMEMultipart()
message['From'] = formataddr(["^_^我是发件人小小明", mail_user])  # 括号里的对应发件人邮箱昵称、发件人邮箱
账号
message['To'] = formataddr(["这里写收件人", receiver])  # 括号里的对应收件人邮箱昵称、收件人邮箱账号
message['Subject'] = Header('Python SMTP 测试', 'utf-8')
# 邮件正文内容
message.attach(MIMEText(mail_msg, 'html', 'utf-8'))

att1 = MIMEText("open('test.txt', 'rb').read()", 'base64', 'utf-8')
att1["Content-Type"] = 'application/octet-stream'
att1["Content-Disposition"] = 'attachment; filename="test.txt"'
message.attach(att1)

fp = open('test.png', 'rb')
bytes = fp.read()
fp.close()

# 构造附件2，传送当前目录下的 runoob.txt 文件
att2 = MIMEImage(bytes)
att2["Content-Type"] = 'application/octet-stream'
att2["Content-Disposition"] = 'attachment; filename="test.png"'
message.attach(att2)

# 定义图片 ID，在 HTML 文本中引用
msgImage = MIMEImage(bytes)
msgImage.add_header('Content-ID', '<image1>')
message.attach(msgImage)

try:
    # smtpObj = smtplib.SMTP()
    # smtpObj.connect(mail_host, 25)  # 25 为 SMTP 端口号
    # smtpObj.login(mail_user, mail_pass)
    # smtpObj.sendmail(sender, receivers, message.as_string())
    server = smtplib.SMTP_SSL("smtp.qq.com", 465)  # 发件人邮箱中的SMTP 服务器，端口是 25
    server.login(mail_user, mail_pass)  # 括号中对应的是发件人邮箱账号、邮箱密码
    server.sendmail(mail_user, receiver, message.as_string())  # 括号中对应的是发件人邮箱账号、收件人邮箱
```

```
账

    print("邮件发送成功")
except smtplib.SMTPException as e:
    print(e)
```

8.6. 多进程

8.6.1. multiprocessing

multiprocessing 是多进程模块，它提供了一个 Process 类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print('Run child process %s (%s)...' % (name, os.getpid()))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Process(target=run_proc, args=('test',))
    print('Child process will start.')
    p.start()
    p.join()
    print('Child process end.')
```

执行结果如下：

```
Parent process 8244.
Child process will start.
Run child process test (6988)...
Child process end.
```

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

8.6.2. Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print('Run task %s (%s)...' % (name, os.getpid()))
    start = time.time()
    time.sleep(random.random() * 3)
```

```
    end = time.time()
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Pool(4)
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print('Waiting for all subprocesses done...')
    p.close()
    p.join()
    print('All subprocesses done.')
```

执行结果如下：

```
Parent process 8076.
Waiting for all subprocesses done...
Run task 0 (8132)...
Run task 1 (8960)...
Run task 2 (8428)...
Run task 3 (7808)...
Task 0 runs 1.74 seconds.
Run task 4 (8132)...
Task 3 runs 2.07 seconds.
Task 1 runs 2.47 seconds.
Task 2 runs 2.64 seconds.
Task 4 runs 2.60 seconds.
All subprocesses done.
```

说明：

对 Pool 对象调用 join()方法会等待所有子进程执行完毕，调用 join()之前必须先调用 close()，调用 close()之后就不能继续添加新的 Process 了。

请注意输出的结果，task 0，1，2，3 是立刻执行的，而 task 4 要等待前面某个 task 完成后才执行，这是因为 `Pool` 的设置为 4，因此，最多同时执行 4 个进程。如果改成：`p = Pool(5)`就可以同时跑 5 个进程。

8.6.3. 进程间通信

Python 的 `multiprocessing` 模块包装了底层的机制，提供了 `Queue`、`Pipes` 等多种方式来交换数据。

我们以 `Queue` 为例，在父进程中创建两个子进程，一个往 `Queue` 里写数据，一个从 `Queue` 里读数据：

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    print('Process to write: %s' % os.getpid())
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())
```

```
# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        if (not value):
            break
        print('Get %s from queue.' % value)

if __name__ == '__main__':
    # 父进程创建Queue，并传给各个子进程：
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程pw，写入:
    pw.start()
    # 启动子进程pr，读取:
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr.terminate()
    #发送结束标记
    q.put(None)
```

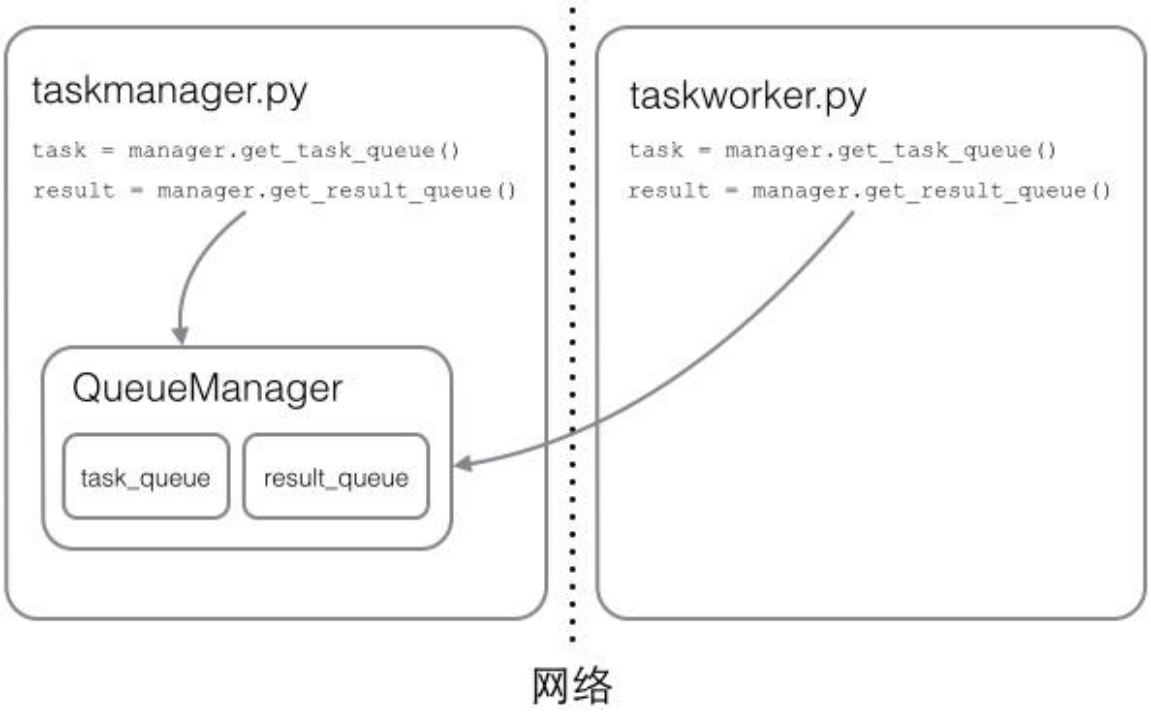
运行结果如下：

```
Process to write: 8152
Put A to queue...
Process to read: 7488
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
Get None from queue.
```

8.6.4. 分布式进程

Python 的 `multiprocessing` 模块支持多进程，其中 `managers` 子模块支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于 `managers` 模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

如果已经有一个通过 `Queue` 通信的多进程程序在同一台机器上运行，通过 `managers` 模块把 `Queue` 通过网络暴露出去，就可以让其他机器的进程访问 `Queue` 了。



我们先看服务进程，服务进程负责启动 `Queue`，把 `Queue` 注册到网络上，然后往 `Queue` 里面写入任务：

```
#coding=utf-8

import queue
import random
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager

# 发送任务的队列:
task_queue = queue.Queue()
def get_task_queue():
    return task_queue
# 接收结果的队列:
result_queue = queue.Queue()
def get_result_queue():
    return result_queue

if __name__ == '__main__':
    freeze_support()

    # 把两个Queue 都注册到网络上, callable 参数关联了Queue 对象:
    BaseManager.register('get_task_queue', callable=get_task_queue)
    BaseManager.register('get_result_queue', callable=get_result_queue)
    # 绑定端口5000, 设置验证码'abc':
    manager = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
    # 启动Queue:
    manager.start()

    # 获得通过网络访问的Queue 对象:
    task = manager.get_task_queue()
```

```
    result = manager.get_result_queue()
    # 放几个任务进去:
    for i in range(10):
        n = random.randint(0, 10000)
        print('Put task %d...' % n)
        task.put(n)
    # 从result队列读取结果:
    print('Try get results...')
    for i in range(10):
        r = result.get(timeout=10)
        print('Result: %s' % r)
    # 关闭:
    manager.shutdown()
```

然后，在另一台机器上启动任务进程（本机上启动也可以）：

```
#coding=utf-8

import queue
import time
from multiprocessing.managers import BaseManager


# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
BaseManager.register('get_task_queue')
BaseManager.register('get_result_queue')

# 连接到服务器，也就是运行taskmanager.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与taskmanager.py设置的完全一致:
m = BaseManager(address=(server_addr, 5000), authkey=b'abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务,并把结果写入result队列:
for i in range(10):
    time.sleep(1)
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = '%d * %d = %d' % (n, n, n*n)
        result.put(r)
    except queue.Empty:
        print('task queue is empty.')

# 处理结束:
print('worker exit.')
```

任务进程要通过网络连接到服务进程，所以要指定服务进程的 IP。

现在，可以试试分布式进程的工作效果了。先启动 taskmanager.py 服务进程：

结果：

```
Put task 8628...
Put task 135...
Put task 1178...
Put task 2009...
Put task 5117...
Put task 1586...
Put task 477...
Put task 3211...
Put task 3681...
Put task 7744...
Try get results...
```

taskmanager 进程发送完任务后，开始等待 result 队列的结果。现在启动 taskworker.py 进程：

```
Connect to server 127.0.0.1...
run task 8628 * 8628...
run task 135 * 135...
run task 1178 * 1178...
run task 2009 * 2009...
run task 5117 * 5117...
run task 1586 * 1586...
run task 477 * 477...
run task 3211 * 3211...
run task 3681 * 3681...
run task 7744 * 7744...
worker exit.
```

taskworker 进程结束，在 taskmanager 进程中会继续打印出结果：

```
Result: 8628 * 8628 = 74442384
Result: 135 * 135 = 18225
Result: 1178 * 1178 = 1387684
Result: 2009 * 2009 = 4036081
Result: 5117 * 5117 = 26183689
Result: 1586 * 1586 = 2515396
Result: 477 * 477 = 227529
Result: 3211 * 3211 = 10310521
Result: 3681 * 3681 = 13549761
Result: 7744 * 7744 = 59969536
```

这个简单的 Manager/Worker 模型就是一个简单但真正的分布式计算，把代码稍加改造，启动多个 worker，就可以把任务分布到几台甚至几十台机器上，比如把计算 n*n 的代码换成发送邮件，就实现了邮件队列的异步发送。

8.7. 多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度。
- 程序的运行速度可能加快。
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可

以释放一些珍贵的资源如内存占用等等。

每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
每个线程都有他自己的一组 CPU 寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的 CPU 寄存器的状态。
指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠） -- 这就是线程的退让。

线程可以分为:

- **内核线程：**由操作系统内核创建和撤销。
- **用户线程：**不需要内核支持而在用户程序中实现的线程。

Python3 线程中常用的两个模块为：

- **_thread**
- **threading(推荐使用)**

thread 模块已被废弃。用户可以使用 threading 模块代替。所以，在 Python3 中不能再使用"thread" 模块。为了兼容性，Python3 将 thread 重命名为 "_thread"。

8.7.1. _thread 创建线程

Python 中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用 _thread 模块中的 start_new_thread()函数来产生新线程。语法如下:

```
_thread.start_new_thread ( function, args[, kwargs] )
```

参数说明:

- function - 线程函数。
- args - 传递给线程函数的参数,他必须是个 tuple 类型。
- kwargs - 可选参数。

实例：

```
import _thread
import time


# 为线程定义一个函数

def print_time(threadName, delay):
    counter = time.perf_counter()
    for count in range(5):
        time.sleep(delay)
        print("线程：%s , 运行时间：%s" % (threadName, round(time.perf_counter() - counter,1)))

# 创建两个线程

try:
```

```
    _thread.start_new_thread(print_time, ("Thread-1", 0.2,))
    _thread.start_new_thread(print_time, ("Thread-2", 0.3,))
except:
    print("Error: 无法启动线程")

while 1:
    pass
```

执行以上程序输出结果如下：

```
线程：Thread-1 ， 运行时间：0.2
线程：Thread-2 ， 运行时间：0.3
线程：Thread-1 ， 运行时间：0.4
线程：Thread-2 ， 运行时间：0.6
线程：Thread-1 ， 运行时间：0.6
线程：Thread-1 ， 运行时间：0.8
线程：Thread-2 ， 运行时间：0.9
线程：Thread-1 ， 运行时间：1.1
线程：Thread-2 ， 运行时间：1.2
线程：Thread-2 ， 运行时间：1.5
```

执行以上程后可以按下 ctrl-c to 退出。

8.7.2. threading 线程模块

threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

- **threading.currentThread()**: 返回当前的线程变量。
- **threading.enumerate()**: 返回一个包含正在运行的线程的 list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- **threading.activeCount()**: 返回正在运行的线程数量，与 len(threading.enumerate())有相同的结果。

除了使用方法外，线程模块同样提供了 Thread 类来处理线程，Thread 类提供了以下方法:

- **run():** 用以表示线程活动的方法。
- **start():**启动线程活动。
- **join([time]):** 等待至线程中止。这阻塞调用线程直至线程的 join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- **isAlive():** 返回线程是否活动的。
- **getName():** 返回线程名。
- **setName():** 设置线程名。

8.7.3. 使用 threading 模块创建线程

我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法：

```
import threading
import time
```

```
class myThread(threading.Thread):
    def __init__(self, name, counter):
        threading.Thread.__init__(self)
        self.name = name
        self.counter = counter

    def run(self):
        print("开始线程：" + self.name)
        print_time()
        print("退出线程：" + self.name)


def print_time():
    counter = time.perf_counter()
    thread = threading.currentThread()
    for count in range(5):
        time.sleep(thread.counter)
        print("线程：%s,counter:%s,运行时间：%s" % (
        thread.name, thread.counter, round(time.perf_counter() - counter, 1)))


# 创建新线程

thread1 = myThread("Thread-1", 0.3)
thread2 = myThread("Thread-2", 0.4)


# 开启新线程

thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("退出主线程")
```

运行结果：

```
开始线程：Thread-1
开始线程：Thread-2
线程：Thread-1,counter:0.3,运行时间：0.3
线程：Thread-2,counter:0.4,运行时间：0.4
线程：Thread-1,counter:0.3,运行时间：0.6
线程：Thread-2,counter:0.4,运行时间：0.8
线程：Thread-1,counter:0.3,运行时间：0.9
线程：Thread-2,counter:0.4,运行时间：1.2
线程：Thread-1,counter:0.3,运行时间：1.2
线程：Thread-1,counter:0.3,运行时间：1.5
退出线程：Thread-1
线程：Thread-2,counter:0.4,运行时间：1.6
线程：Thread-2,counter:0.4,运行时间：2.0
退出线程：Thread-2
退出主线程
```

8.7.4. Lock 线程同步

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

```
import threading
import time


class myThread(threading.Thread):
    threadLock = threading.Lock()

    def __init__(self, name, delay):
        threading.Thread.__init__(self)
        self.name = name
        self.delay = delay

    def run(self):
        print("开始线程：" + self.name)

        # 获取锁，用于线程同步

        myThread.threadLock.acquire()
        print_time()

        # 释放锁，开启下一个线程

        myThread.threadLock.release()
        print("退出线程：" + self.name)


def print_time():
    counter = time.perf_counter()
    thread = threading.currentThread()
    for count in range(3):
        time.sleep(thread.delay)
        print("线程：%s,counter:%s,运行时间：%s" % (
            thread.name, thread.delay, round(time.perf_counter() - counter, 1)))


# 创建新线程

thread1 = myThread("Thread-1", 0.3)
thread2 = myThread("Thread-2", 0.4)

# 开启新线程

thread1.start()
thread2.start()
thread1.join()
```

```
thread2.join()
print("退出主线程")
```

执行以上程序，输出结果为：

```
开始线程：Thread-1
开始线程：Thread-2
线程：Thread-1,counter:0.3,运行时间：0.3
线程：Thread-1,counter:0.3,运行时间：0.6
线程：Thread-1,counter:0.3,运行时间：0.9
退出线程：Thread-1
线程：Thread-2,counter:0.4,运行时间：0.4
线程：Thread-2,counter:0.4,运行时间：0.8
线程：Thread-2,counter:0.4,运行时间：1.2
退出线程：Thread-2
退出主线程
```

```
import threading

# 假定这是你的银行存款:
balance = 0
lock = threading.Lock()

def change_it(n):
    # 先存后取，结果应该为0:
    global balance
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

当多个线程同时执行 lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用 try...finally 来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

8.7.5. 线程优先级队列（Queue）

Python 的 Queue 模块中提供了同步的、线程安全的队列类，包括 FIFO（先入先出)队列 Queue，LIFO（后入先出）队列 LifoQueue，和优先级队列 PriorityQueue。
这些队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步。
Queue 模块中的常用方法:

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回 True,反之 False
- Queue.full() 如果队列满了，返回 True,反之 False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout 等待时间
- Queue.get_nowait() 相当 Queue.get(False)
- Queue.put(item) 写入队列，timeout 等待时间
- Queue.put_nowait(item) 相当 Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

生产者消费者实例:

```
import queue
import random
import threading
import time


class Producer(threading.Thread):
    nameList = ["apple", "peach", "pineapple", "orange", "banana", "blueberry"]
    flag = 1

    def __init__(self, q, name):
        threading.Thread.__init__(self)
        self.name = name
        self.q = q

    def run(self):
        name_list = Producer.nameList
        while Producer.flag:
            queueLock.acquire()
            if not self.q.full():
                data = name_list[random.randrange(0, len(name_list))]
                self.q.put(data)
                print("%s 生产数据: %s" % (threading.currentThread().name, data))
                queueLock.release()
            else:
                queueLock.release()
```

```
            time.sleep(random.random() * 3)


class Consumer(threading.Thread):
    flag = 1

    def __init__(self, q, name):
        threading.Thread.__init__(self)
        self.name = name
        self.q = q

    def run(self):
        while Consumer.flag:
            queueLock.acquire()
            if not self.q.empty():
                data = self.q.get()
                print("%s 消费数据: %s" % (threading.currentThread().name, data))
                queueLock.release()
            else:
                queueLock.release()
            time.sleep(random.random() * 4)


workQueue = queue.Queue(5)
queueLock = threading.Lock()

# 创建新线程

Producer(workQueue, "Producer1").start()
Producer(workQueue, "Producer2").start()
Consumer(workQueue, "Consumer1").start()
Consumer(workQueue, "Consumer2").start()
Consumer(workQueue, "Consumer3").start()

while 1:
    time.sleep(1)
    print(workQueue.queue)
```

8.7.6. Python 线程的 GIL 锁

正常情况下，如果有两个死循环线程，在多核 CPU 中，可以监控到会占用 200%的 CPU，也就是占用两个 CPU 核心。要想把 N 核 CPU 的核心全部跑满，就必须启动 N 个死循环线程。

试试用 Python 写个死循环：

```
import threading, multiprocessing
def loop():
    x = 0
    while True:
        x = x ^ 1
for i in range(multiprocessing.cpu_count()):
```

```
    t = threading.Thread(target=loop)
    t.start()
```

启动与 CPU 核心数量相同的 N 个线程，在 4 核 CPU 上可以监控到 CPU 占用率仅有 160%，也就是使用不到两核。
即使启动 100 个线程，使用率也就 170%左右，仍然不到两核。
但是用 C、C++或 Java 来改写相同的死循环，直接可以把全部核心跑满，4 核就跑到 400%，8 核就跑到 800%，为什么 Python 不行呢？
因为 Python 的线程虽然是真正的线程，但解释器执行代码时，有一个 GIL 锁：Global Interpreter Lock，任何 Python 线程执行前，必须先获得 GIL 锁，然后，每执行 100 条字节码，解释器就自动释放 GIL 锁，让别的线程有机会执行。这个 GIL 全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在 Python 中只能交替执行，即使 100 个线程跑在 100 核 CPU 上，也只能用到 1 个核。
GIL 是 Python 解释器设计的历史遗留问题，通常我们用的解释器是官方实现的 CPython，要真正利用多核，除非重写一个不带 GIL 的解释器。
所以，在 Python 中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过 C 扩展来实现，不过这样就失去了 Python 简单易用的特点。
不过，也不用过于担心，Python 虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个 Python 进程有各自独立的 GIL 锁，互不影响。
多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。
Python 解释器由于设计时有 GIL 全局锁，导致了多线程无法利用多核无法实现多线程的并发

8.7.7. ThreadLocal

ThreadLocal 是一个以 thread 自身作为 key 的全局 dict：

```
import threading

# 创建全局ThreadLocal对象:
local_school = threading.local()

def process_student():
    # 获取当前线程关联的student:
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    # 绑定ThreadLocal的student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=('Alice',), name='Thread-A')
t2 = threading.Thread(target= process_thread, args=('Bob',), name='Thread-B')
t1.start()
t2.start()
t1.join()
t2.join()
```

执行结果：

```
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

全局变量 `local_school` 就是一个 `ThreadLocal` 对象，每个 `Thread` 对它都可以读写 `student` 属性，但互不影响。你可以把 `local_school` 看成全局变量，但每个属性如 `local_school.student` 都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，`ThreadLocal` 内部会处理。

可以理解为全局变量 `local_school` 是一个 `dict`，不但可以用 `local_school.student`，还可以绑定其他变量，如 `local_school.teacher` 等等。

`ThreadLocal` 最常用的地方就是为每个线程绑定一个数据库连接，HTTP 请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

8.8. 异步 IO

CPU 的速度远远快于磁盘、网络等 IO。在一个线程中，CPU 执行代码的速度极快，然而，一旦遇到 IO 操作，如读写文件、发送网络数据时，就需要等待 IO 操作完成，才能继续进行下一步操作。这种情况称为同步 IO。
在 IO 操作的过程中，当前线程被挂起，而其他需要 CPU 执行的代码就无法被当前线程执行了。
因为一个 IO 操作就阻塞了当前线程，导致其他代码无法执行，所以我们必须使用多线程或者多进程来并发执行代码，为多个用户服务。每个用户都会分配一个线程，如果遇到 IO 导致线程被挂起，其他用户的线程不受影响。
多线程和多进程的模型虽然解决了并发问题，但是系统不能无上限地增加线程。由于系统切换线程的开销也很大，所以，一旦线程数量过多，CPU 的时间就花在线程切换上了，真正运行代码的时间就少了，结果导致性能严重下降。
由于我们要解决的问题是 CPU 高速执行能力和 IO 设备的龟速严重不匹配，多线程和多进程只是解决这一问题的一种方法。
另一种解决 IO 问题的方法是异步 IO。当代码需要执行一个耗时的 IO 操作时，它只发出 IO 指令，并不等待 IO 结果，然后就去执行其他代码了。一段时间后，当 IO 返回结果时，再通知 CPU 进行处理。

异步 IO 模型需要一个消息循环，在消息循环中，主线程不断地重复“读取消息-处理消息”这一过程：

```
loop = get_event_loop()
while True:
    event = loop.get_event()
    process_event(event)
```

消息模型其实早在应用在桌面应用程序中了。一个 GUI 程序的主线程就负责不停地读取消息并处理消息。所有的键盘、鼠标等消息都被发送到 GUI 程序的消息队列中，然后由 GUI 程序的主线程处理。
由于 GUI 线程处理键盘、鼠标等消息的速度非常快，所以用户感觉不到延迟。某些时候，GUI 线程在一个消息处理的过程中遇到问题导致一次消息处理时间过长，此时，用户会感觉到整个 GUI 程序停止响应了，敲键盘、点鼠标都没有反应。这种情况说明在消息模型中，处理一个消息必须非常迅速，否则，主线程将无法及时处理消息队列中的其他消息，导致程序看上去停止响应。
消息模型是如何解决同步 IO 必须等待 IO 操作这一问题的呢？当遇到 IO 操作时，代码只负责发出 IO 请求，不等待 IO 结果，然后直接结束本轮消息处理，进入下一轮消息处理过程。当 IO 操作完成后，将收到一条“IO 完成”的消息，处理该消息时就可以直接获取 IO 操作结果。
在“发出 IO 请求”到收到“IO 完成”的这段时间里，同步 IO 模型下，主线程只能挂起，但异步 IO 模型下，主线程并没有休息，而是在消息循环中继续处理其他消息。这样，在异步 IO 模型下，一个线程就可以同时处理多个 IO 请求，并且没有切换线程的操作。对于大多数 IO 密集型的应用程序，使用异步 IO 将大大提升系统的多任务处理能力。

8.8.1. 协程

协程，又称微线程，纤程。英文名 Coroutine。
子程序，或者称为函数，在所有语言中都是层级调用，比如 A 调用 B，B 在执行过程中又调用了 C，C 执行完毕返回，B 执行完毕返回，最后是 A 执行完毕。

所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。
子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。
协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。
注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似 CPU 的中断。比如子程序 A、B：

```
def A():
    print('1')
    print('2')
    print('3')
def B():
    print('x')
    print('y')
    print('z')
```

假设由协程执行，在执行 A 的过程中，可以随时中断，去执行 B，B 也可能在执行过程中中断再去执行 A，结果可能是：

```
1
2
x
y
3
z
```

但是在 A 中是没有调用 B 的，所以协程的调用比函数调用理解起来要难一些。
看起来 A、B 的执行有点像多线程，但协程的特点在于是一个线程执行,子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。
只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。
因为协程是一个线程执行，那怎么利用多核 CPU 呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python 对协程的支持是通过 generator 实现的。
在 generator 中，我们不但可以通过 for 循环来迭代，还可以不断调用 next()函数获取由 yield 语句返回的下一个值。
但是 Python 的 yield 不但可以返回一个值，它还可以接收调用者发出的参数。
来看例子：
传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。
如果改用协程，生产者生产消息后，直接通过 yield 跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

```
import random


def consumer():
    r = "consumer starting"
    while True:
        data = yield r
        print('[CONSUMER] Consuming %s...' % data)
        r = '200 OK'

def produce(c):
    r = c.send(None)
    name_list = ["apple", "peach", "pineapple", "orange", "banana", "blueberry"]
    for i in range(5):
```

```
        data = name_list[random.randrange(0, len(name_list))]
        print("[PRODUCER] Producing %s" % data)
        r = c.send(data)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()


c = consumer()
produce(c)
```

执行结果：

```
[PRODUCER] Producing banana
[CONSUMER] Consuming banana...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing banana
[CONSUMER] Consuming banana...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing apple
[CONSUMER] Consuming apple...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing pineapple
[CONSUMER] Consuming pineapple...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing pineapple
[CONSUMER] Consuming pineapple...
[PRODUCER] Consumer return: 200 OK
```

注意到 consumer 函数是一个 generator，把一个 consumer 传入 produce 后：

首先调用 c.send(None)启动生成器；
然后，一旦生产了东西，通过 c.send(data)切换到 consumer 执行；
consumer 通过 yield 拿到消息，处理，又通过 yield 把结果传回；
produce 拿到 consumer 处理的结果，继续生产下一条消息；
produce 决定不生产了，通过 c.close()关闭 consumer，整个过程结束。
整个流程无锁，由一个线程执行，produce 和 consumer 协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

8.8.2. asyncio

asyncio 是 Python 3.4 版本引入的标准库，直接内置了对异步 IO 的支持。
asyncio 的编程模型就是一个消息循环。我们从 asyncio 模块中直接获取一个 EventLoop 的引用，然后把需要执行的协程扔到 EventLoop 中执行，就实现了异步 IO。
用 asyncio 实现 Hello world 代码如下：

```
import asyncio

@asyncio.coroutine
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")
```

```
# 获取EventLoop:
loop = asyncio.get_event_loop()# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

`@asyncio.coroutine`把一个 generator 标记为 coroutine 类型，然后就可以把这个 `coroutine`扔到 `EventLoop`中执行。

`yield from`语法可以让我们方便地调用另一个 `generator`。由于 `asyncio.sleep()`也是一个 `coroutine`，所以线程不会等待 `asyncio.sleep()`，而是直接中断并执行下一个消息循环。当 `asyncio.sleep()`返回时，线程就可以从 `yield from`拿到返回值（此处是 `None`），然后接着执行下一行语句。

把 `asyncio.sleep(1)`看成是一个耗时 1 秒的 IO 操作，在此期间，主线程并未等待，而是去执行 `EventLoop`中其他可以执行的 `coroutine`了，因此可以实现并发执行。

用 Task 封装两个 `coroutine`：

```
import threading
import asyncio

@asyncio.coroutine
def hello():
    print('Hello world! (%s)' % threading.currentThread())
    yield from asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

观察执行过程：

```
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
Hello world! (<_MainThread(MainThread, started 140735195337472)>)
(暂停约 1 秒)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
Hello again! (<_MainThread(MainThread, started 140735195337472)>)
```

由打印的当前线程名称可以看出，两个 coroutine 是由同一个线程并发执行的。

如果把 asyncio.sleep()换成真正的 IO 操作，则多个 coroutine 就可以由一个线程并发执行。

我们用 asyncio 的异步网络连接来获取 sina、sohu 和 163 的网站首页：

```
import asyncio

@asyncio.coroutine
def wget(host):
    print('wget %s...' % host)
    connect = asyncio.open_connection(host, 80)
    reader, writer = yield from connect
    header = \
"""GET / HTTP/1.0
Host: %s

""" % host
    writer.write(header.encode('utf-8'))
```

```
    yield from writer.drain()
    while True:
        line = yield from reader.readline()
        if line == b'\r\n':
            break
        print('%s header > %s' % (host, line.decode('utf-8').rstrip()))
    # Ignore the body, close the socket
    writer.close()

loop = asyncio.get_event_loop()
tasks = [wget(host) for host in ['www.sina.com.cn', 'www.sohu.com', 'www.163.com']]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

asyncio 提供了完善的异步 IO 支持；
异步操作需要在 coroutine 中通过 yield from 完成；
多个 coroutine 可以封装成一组 Task 然后并发执行。

8.8.3. async/await

从 Python 3.5 开始引入了新的语法 async 和 await，可以让 coroutine 的代码更简洁易读。
请注意，async 和 await 是针对 coroutine 的新语法，要使用新的语法，只需要做两步简单的替换：

- 把@asyncio.coroutine 替换为 async；
- 把 yield from 替换为 await。

上面的例子可改写为：

```
import threading
import asyncio

async def hello():
    print('Hello world! (%s)' % threading.currentThread())
    await asyncio.sleep(1)
    print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get_event_loop()
tasks = [hello(), hello()]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

```
import asyncio

async def wget(host):
    print('wget %s...' % host)
    connect = asyncio.open_connection(host, 80)
    reader, writer = await connect
    header = \
"""GET / HTTP/1.0
Host: %s

""" % host
    writer.write(header.encode('utf-8'))
```

```
    await writer.drain()
    while True:
        line = await reader.readline()
        if line == b'\r\n':
            break
        print('%s header > %s' % (host, line.decode('utf-8').rstrip()))
    # Ignore the body, close the socket
    writer.close()

loop = asyncio.get_event_loop()
tasks = [wget(host) for host in ['www.sina.com.cn', 'www.sohu.com', 'www.163.com']]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

8.9. Python3 JSON 数据解析

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于 ECMAScript 的一个子集。
Python3 中可以使用 json 模块来对 JSON 数据进行编解码，它包含了两个函数：

- **json.dumps():** 对数据进行编码。
- **json.loads():** 对数据进行解码。

在 json 的编解码过程中，python 的原始类型与 json 类型会相互转换，具体的转化对照如下：
Python 编码为 JSON 类型转换对应表：

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
TRUE	TRUE
FALSE	FALSE
None	null

JSON 解码为 Python 类型转换对应表：

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
TRUE	TRUE
FALSE	FALSE
null	None

8.9.1. json.dumps

以下实例演示了 Python 数据结构转换为 JSON：

```
import json


# Python 字典类型转换为 JSON 对象

data = {
    'no': 1,
    'name': 'Baidu',
    'url': 'http://www.baidu.com'
}

json_str = json.dumps(data)
print("Python 原始数据：", data)
print("JSON 对象：", json_str)
```

执行以上代码输出结果为：

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

8.9.2. json.loads

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

```
import json


# Python 字典类型转换为 JSON 对象

data1 = {
    'no': 1,
    'name': 'Baidu',
    'url': 'http://www.baidu.com'
}

json_str = json.dumps(data1)
print("Python 原始数据：", data1)
print("JSON 对象：", json_str)


# 将 JSON 对象转换为 Python 字典

data2 = json.loads(json_str)
print(type(data2), data2)
```

执行以上代码输出结果为：

```
Python 原始数据： {'no': 1, 'name': 'Baidu', 'url': 'http://www.baidu.com'}
JSON 对象： {"no": 1, "name": "Baidu", "url": "http://www.baidu.com"}
<class 'dict'> {'no': 1, 'name': 'Baidu', 'url': 'http://www.baidu.com'}
```

8.9.3. json.dump() 和 json.load()

json.dump()可将数据编码到文件中，json.load()可加载 JSON 文件并解析出 json 对象。例如：

```
import json

data = {
    'no': 1,
    'name': 'Baidu',
    'url': 'http://www.baidu.com'
}

with open('data.json', 'w') as f:
    json.dump(data, f)


# 读取数据

with open('data.json', 'r') as f:
    data = json.load(f)
    print(data)
```

8.10. 时间&日期

8.10.1. Time 模块内置函数

函数	描述
time.altzone	返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。对夏令时启用地区才能使用。
time.asctime([tupletime])	接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008 年 12 月 11 日 周二 18 时 07 分 14 秒）的 24 个字符的字符串。
time.perf_counter() time.process_time()	用以浮点数计算的秒数返回当前的 CPU 时间。用来衡量不同程序的耗时，比 time.time()更有用。
time.ctime([secs])	作用相当于 asctime(localtime(secs))，未给参数相当于 asctime()
time.gmtime([secs])	接收时间辍（1970 纪元后经过的浮点秒数）并返回格林威治天文时间下的时间元组 t。注：t.tm_isdst 始终为 0
time.localtime([secs])	接收时间辍（1970 纪元后经过的浮点秒数）并返回当地时间下的时间元组 t（t.tm_isdst 可取 0 或 1，取决于当地当时是不是夏令时）。
time.mktime(tupletime)	接受时间元组并返回时间辍（1970 纪元后经过的浮点秒数）。
time.sleep(secs)	推迟调用线程的运行，secs 指秒数。

time.strftime(fmt[,tupletime])	接收以时间元组，并返回以可读字符串表示的当地时间，格式由 fmt 决定。
time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')	根据 fmt 的格式把一个时间字符串解析为时间元组。
time.time( )	返回当前时间的时间戳（1970 纪元后经过的浮点秒数）。
time.tzset()	根据环境变量 TZ 重新初始化时间相关设置。

8.10.2. perf_counter 进度条实例

```
print(time.perf_counter())  # 返回系统运行时间

print(time.process_time())  # 返回进程运行时间
```

```
import time

scale = 30

print("执行开始".center(scale//2,"-"))  # .center() 控制输出的样式，宽度为 25//2，即 22，汉字居中，两侧填充 -


start = time.perf_counter() # 调用一次 perf_counter()，从计算机系统里随机选一个时间点 A，计算其距离当前时间点 B1 有多少秒。当第二次调用该函数时，默认从第一次调用的时间点 A 算起，距离当前时间点 B2 有多少秒。两个函数取差，即实现从时间点 B1 到 B2 的计时功能。
for i in range(scale+1):
    a = '*' * i             # i 个长度的 * 符号
    b = '.' * (scale-i)  # scale-i ）个长度的 . 符号。符号 * 和 . 总长度为 50
    c = (i/scale)*100  # 显示当前进度，百分之多少
    dur = time.perf_counter() - start    # 计时，计算进度条走到某一百分比的用时
    print("\r{:^3.0f}%[{}->{}]{:.2f}s".format(c,a,b,dur),end='')  # \r 用来在每次输出完成后，将光标移至行首，这样保证进度条始终在同一行输出，即在一行不断刷新的效果；{:^3.0f}，输出格式为居中，占 3 位，小数点后 0 位，浮点型数，对应输出的数为 c；{}，对应输出的数为 a；{}，对应输出的数为 b；{:.2f}，输出有两位小数的浮点数，对应输出的数为 dur；end=''，用来保证不换行，不加这句默认换行。
    time.sleep(0.1)     # 在输出下一个百分之几的进度前，停止 0.1 秒
```

```
print("\n"+"执行结果".center(scale//2,'-'))
```

8.10.3. struct_time 元组属性

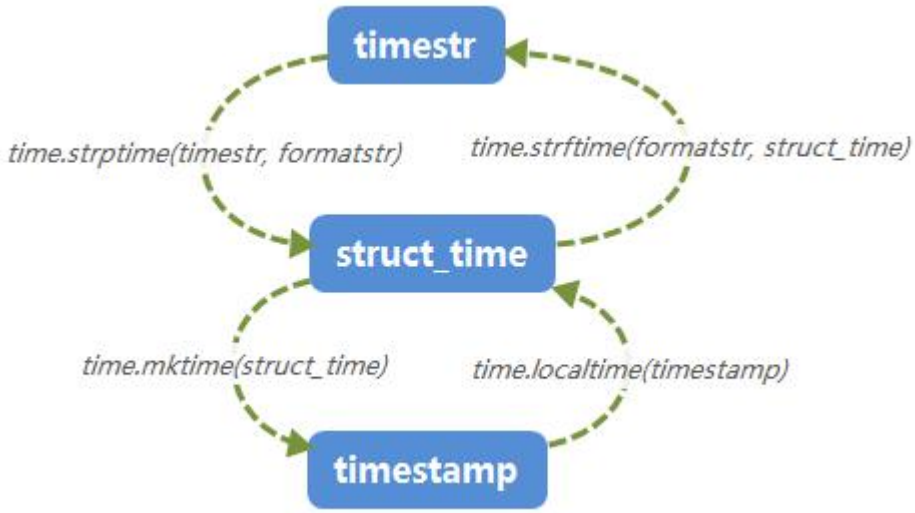
属性	字段	值
tm_year	4 位数年	2008
tm_mon	月	1 到 12
tm_mday	日	1 到 31
tm_hour	小时	0 到 23
tm_min	分钟	0 到 59
tm_sec	秒	0 到 61 (60 或 61 是闰秒)
tm_wday	一周的第几日	0 到 6 (0 是周日)
tm_yday	一年的第几日	1 到 366 (儒略历)
tm_isdst	夏令时	-1, 0, 1, -1 是决定是否为夏令时的旗帜

8.10.4. 时间日期格式化符号

符号	含义	示例
%y	两位数的年份表示（00-99）	17
%Y	四位数的年份表示（000-9999）	2017
%m	月份（01-12）	08
%d	月内中的一天（0-31）	27
%H	24 小时制小时数（0-23）	11
%I	12 小时制小时数（01-12）	11
%M	分钟数（00=59）	29
%S	秒（00-59）	08
%a	本地简化星期名称	Sun
%A	本地完整星期名称	Sunday
%b	本地简化的月份名称	Aug
%B	本地完整的月份名称	August
%c	本地相应的日期表示和时间表示	08/27/17 11:29:08
%j	年内的一天（001-366）	239
%p	本地 A.M.或 P.M.的等价符	AM

%U	一年中的星期数（00-53）星期天为星期的开始	35
%w	星期（0-6），星期天为星期的开始	0
%W	一年中的星期数（00-53）星期一为星期的开始	34
%x	本地相应的日期表示	08/27/17
%X	本地相应的时间表示	11:29:08
%Z	当前时区的名称	中国标准时间
%%	%号本身	%

8.10.5. 时间字符串与 struct_time 元组与时间戳间的相互转换




```
#获取当前时间戳
time.time()
#将指定时间戳转换为 struct_time 元组
time.localtime(timestamp)
#将指定 struct_time 元组转换为指定格式的字符串
time.strftime(formatstr, struct_time)
#将指定格式的字符串解析为 struct_time 元组
time.strptime(timestr, formatstr)
#将 struct_time 元组转换为时间戳
time.mktime(struct_time)
```

8.10.5.1. 获取当前时间戳

```
import time  # 引入 time 模块

timecur = time.time()
print("当前时间戳为", timecur, type(timecur))
```

8.10.5.2. 时间戳转换为 struct_time 元组

```
localtime = time.localtime(1459175064.0)
print("本地时间为 :", localtime, type(localtime))
```

8.10.5.3. struct_time 元组转换为时间字符串

```
print(time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(1459175064.0)))
```

8.10.5.4. 时间字符串转换为 struct_time 元组

```
strptime = time.strptime("2016-03-28 22:24:24", "%Y-%m-%d %H:%M:%S")
print(strptime, type(strptime))
```

8.10.5.5. struct_time 元组转换为时间戳

```
print(time.mktime(time.localtime(1459175064.0)))
```

结果：

1459175064.0

8.10.6. datetime 模块

datetime 是 Python 处理日期和时间的标准库。

datetime 是模块，datetime 模块还包含一个 datetime 类，通过 from datetime import datetime 导入的才是 datetime 这个类。

如果仅导入 import datetime，则必须引用全名 datetime.datetime。

8.10.6.1. 获取当前/指定日期和时间

```
from datetime import datetime

# 获取当前 datetime:
now = datetime.now()
print('now =', now)
print('type(now) =', type(now))

# 用指定日期时间创建 datetime:
dt = datetime(2015, 4, 19, 12, 20)
print('dt =', dt)
```

datetime.now()返回当前日期和时间，其类型是 datetime。

8.10.6.2. datetime 转换为 timestamp

在计算机中，时间实际上是用数字表示的。我们把 1970 年 1 月 1 日 00:00:00 UTC+00:00 时区的时刻称为 epoch time，记为 0（1970 年以前的时间 timestamp 为负数），当前时间就是相对于 epoch time 的秒数，称为 timestamp。

可以认为：

```
timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00
```

对应的北京时间是：

```
timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00
```

可见 timestamp 的值与时区毫无关系，因为 timestamp 一旦确定，其 UTC 时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以 timestamp 表示的，因为全球各地的计算机在任意时刻的 timestamp 都是完全相同的（假定时间已校准）。

把一个 datetime 类型转换为 timestamp 只需要简单调用 timestamp()方法：

```
from datetime import datetime
dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建 datetime
dt.timestamp() # 把 datetime 转换为 timestamp
```

注意:Python 的 timestamp 是一个浮点数。如果有小数位，小数位表示毫秒数。
某些编程语言（如 Java 和 JavaScript）的 timestamp 使用整数表示毫秒数，这种情况下只需要把 timestamp 除以 1000 就得到 Python 的浮点表示方法。

8.10.6.3. timestamp 转换为 datetime

要把 timestamp 转换为 `datetime`，使用 `datetime` 提供的 `fromtimestamp()` 方法：

```
from datetime import datetime
t = 1429417200.0
print(datetime.fromtimestamp(t))
```

注意到 timestamp 是一个浮点数，它没有时区的概念，而 datetime 是有时区的。上述转换是在 timestamp 和本地时间做转换。
本地时间是指当前操作系统设定的时区。例如北京时区是东 8 区，则本地时间：

```
2015-04-19 12:20:00
```

实际上就是 UTC+8:00 时区的时间：

```
2015-04-19 12:20:00 UTC+8:00
```

而此刻的格林威治标准时间与北京时间差了 8 小时，也就是 UTC+0:00 时区的时间应该是：

```
2015-04-19 04:20:00 UTC+0:00
```

timestamp 也可以直接被转换到 UTC 标准时区的时间：

```
from datetime import datetime
t = 1429417200.0
print(datetime.fromtimestamp(t)) # 本地时间
# 2015-04-19 12:20:00
print(datetime.utcfromtimestamp(t)) # UTC 时间
# 2015-04-19 04:20:00
```

8.10.6.4. str 转换为 datetime

```
from datetime import datetime
cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')
print(cday)
# 2015-06-01 18:19:59
```

8.10.6.5. datetime 转换为 str

```
from datetime import datetime
now = datetime.now()
print(now.strftime('%a, %b %d %H:%M'))
```

8.10.6.6. datetime 加减

```
from datetime import datetime, timedelta
now = datetime.now()
print(now)
```

```
# 2019-10-04 16:43:32.744069
print(now + timedelta(hours=10))
# 2019-10-05 02:43:32.744069
print(now - timedelta(days=1))
# 2019-10-03 16:43:32.744069
print(now + timedelta(days=2, hours=12))
# 2019-10-07 04:43:32.744069
```

可见，使用 timedelta 你可以很容易地算出前几天和后几天的时刻。

8.10.6.7. 设置 UTC 时区

本地时间是指系统设定时区的时间，例如北京时间是 UTC+8:00 时区的时间，而 UTC 时间指 UTC+0:00 时区的时间。

一个 `datetime` 类型有一个时区属性 `tzinfo`，但是默认为 `None`，所以无法区分这个 `datetime` 到底是哪个时区，除非给 `datetime` 设置一个时区：

```
from datetime import datetime, timedelta, timezone
tz_utc_8 = timezone(timedelta(hours=8)) # 创建时区 UTC+8:00
now = datetime.now()
dt = now.replace(tzinfo=tz_utc_8) # 设置时区为UTC+8:00
```

如果系统时区恰好是 UTC+8:00，那么上述代码就是正确的，否则，不能设置为 UTC+8:00 时区。

8.10.6.8. 时区转换

```
from datetime import datetime, timedelta, timezone

# 拿到UTC 时间，并强制设置时区为UTC+0:00:
utc_dt = datetime.utcnow().replace(tzinfo=timezone.utc)
print(utc_dt)
# astimezone()将转换时区为北京时间:
bj_dt = utc_dt.astimezone(timezone(timedelta(hours=8)))
print(bj_dt)
# astimezone()将转换时区为东京时间:
tokyo_dt = utc_dt.astimezone(timezone(timedelta(hours=9)))
print(tokyo_dt)
# astimezone()将 bj_dt 转换时区为东京时间:
tokyo_dt2 = bj_dt.astimezone(timezone(timedelta(hours=9)))
print(tokyo_dt2)
```

8.10.6.9. 将指定时区的时间字符串转换为时间戳

```
# 假设你获取了用户输入的日期和时间如2015-1-21 9:01:30，以及一个时区信息如UTC+5:00，均是 str，
# 请编写一个函数将其转换为timestamp:
def to_timestamp(dt_str, tz_str):
    offesthour = int(re.match(r"UTC([-+]\d+):00", tz_str).group(1))
    strptime = datetime.strptime(dt_str, '%Y-%m-%d %H:%M:%S')
    astimezone = strptime.replace(tzinfo=timezone(timedelta(hours=offesthour)))
    return astimezone.timestamp()

t1 = to_timestamp('2015-6-1 08:10:30', 'UTC+7:00')
assert t1 == 1433121030.0

t2 = to_timestamp('2015-5-31 16:10:30', 'UTC-09:00')
assert t2 == 1433121030.0
```

完整示例：

```
import re
from datetime import datetime, timedelta, timezone

# 获取当前 datetime:
now = datetime.now()
print('now =', now)
print('type(now) =', type(now))

# 用指定日期时间创建 datetime:
dt = datetime(2020, 4, 19, 12, 20)
print('dt =', dt)

# 把 datetime 转换为 timestamp:
print('datetime -> timestamp:', dt.timestamp())

# 把 timestamp 转换为 datetime:
t = dt.timestamp()
print('timestamp -> datetime:', datetime.fromtimestamp(t))
print('timestamp -> datetime as UTC+0:', datetime.utcfromtimestamp(t))

# 从 str 读取 datetime:
cday = datetime.strptime('2020-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')
print('strptime:', cday)

# 把 datetime 格式化输出:
print('strftime:', cday.strftime('%a, %b %d %H:%M'))

# 对日期进行加减:
print('current datetime =', cday)
print('current + 10 hours =', cday + timedelta(hours=10))
print('current - 1 day =', cday - timedelta(days=1))
print('current + 2.5 days =', cday + timedelta(days=2, hours=12))

# 把时间从 UTC+0 时区转换为 UTC+8:
utc_dt = datetime.utcnow().replace(tzinfo=timezone.utc)
utc8_dt = utc_dt.astimezone(timezone(timedelta(hours=8)))
print('UTC+0:00 now =', utc_dt)
print('UTC+8:00 now =', utc8_dt)


# 假设你获取了用户输入的日期和时间如 2015-1-21 9:01:30，以及一个时区信息如 UTC+5:00，均是 str，
# 请编写一个函数将其转换为 timestamp:
def to_timestamp(dt_str, tz_str):
    offesthour = int(re.match(r"UTC([-+]\d+):00", tz_str).group(1))
    strptime = datetime.strptime(dt_str, '%Y-%m-%d %H:%M:%S')
    astimezone = strptime.replace(tzinfo=timezone(timedelta(hours=offesthour)))
    return astimezone.timestamp()
```

```
t1 = to_timestamp('2015-6-1 08:10:30', 'UTC+7:00')
assert t1 == 1433121030.0
t2 = to_timestamp('2015-5-31 16:10:30', 'UTC-09:00')
assert t2 == 1433121030.0
print('ok')
```

8.10.7. 日历（Calendar）模块

函数	描述
calendar.calendar(year,w=2,l=1,c=6)	返回一个多行字符串格式的 year 年年历，3 个月一行，间隔距离为 c。 每日宽度间隔为 w 字符。每行长度为 21* W+18+2* C。l 是每星期行数。
calendar.setfirstweekday(weekday)	设置每周的起始日期码。0（星期一）到 6（星期日）。
calendar.firstweekday( )	返回当前每周起始日期的设置。默认情况下，首次载入 caendar 模块时返回 0，即星期一。
calendar.timegm(tupletime)	和 time.gmtime 相反：接受一个时间元组形式，返回该时刻的时间戳（1970 纪元后经过的浮点秒数）。
calendar.isleap(year)	是闰年返回 True，否则为 false。
calendar.leapdays(y1,y2)	返回在 Y1，Y2 两年之间的闰年总数。
calendar.month(year,month,w=2,l=1)	返回一个多行字符串格式的 year 年 month 月日历，两行标题，一周一行。每日宽度间隔为 w 字符。每行的长度为 7* w+6。l 是每星期的行数。
calendar.monthcalendar(year,month)	返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。Year 年 month 月外的日期都设为 0;范围内的日子都由该月第几日表示，从 1 开始。
calendar.monthrange(year,month)	返回两个整数。第一个是该月的星期几，第二个是该月有几天。星期几是从 0（星期一）到 6（星期日）。
	calendar.monthrange(2014, 11)(5, 30)解释：5 表示 2014 年 11 月份的第一天是周六，30 表示 2014 年 11 月份总共有 30 天。
calendar.prcal(year,w=2,l=1,c=6)	相当于 print(calendar.calendar(year,w,l,c))
calendar.prmonth(year,month,w=2,l=1)	相当于 print(calendar.month(year,month,w,l))
calendar.weekday(year,month,day)	返回给定日期的日期码。0（星期一）到 6（星期日）。月份为 1（一月）到 12（12 月）。

```
import calendar

# 输入指定年月
yy, mm = 18, 10
# 显示日历
print(calendar.month(yy, mm))
```

```
    October 18
Mo Tu We Th Fr Sa Su
```

```
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

8.11. collections

collections 是 Python 内建的一个集合模块，提供了许多有用的集合类。

8.11.1. namedtuple

`namedtuple` 是一个函数，它用来创建一个自定义的 `tuple` 对象，并且规定了 `tuple` 元素的个数，并可以用属性而不是索引来引用 `tuple` 的某个元素。
这样一来，我们用 `namedtuple` 可以很方便地定义一种数据类型，它具备 `tuple` 的不变性，又可以根据属性来引用，使用十分方便。

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(1, 2)
print('Point:', p.x, p.y)
```

可以验证创建的 Point 对象是 tuple 的一种子类：

```
print(isinstance(p, Point), isinstance(p, tuple))
#True True
```

类似的，如果要用坐标和半径表示一个圆，也可以用 `namedtuple` 定义：

```
# namedtuple('名称', [属性list]):
Circle = namedtuple('Circle', ['x', 'y', 'r'])
```

8.11.2. deque

使用 list 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 list 是线性存储，数据量大的时候，插入和删除效率很低。
deque 是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

```
from collections import deque

q = deque(['a', 'b', 'c'])
q.append('x')
q.appendleft('y')
print(q)
```

8.11.3. defaultdict

使用 dict 时，如果引用的 Key 不存在，就会抛出 KeyError。如果希望 key 不存在时，返回一个默认值，就可以用 defaultdict：

```
from collections import defaultdict
```

```
dd = defaultdict(lambda: 'N/A')
dd['key1'] = 'abc'
print('dd[\'key1\'] =', dd['key1'])
print('dd[\'key2\'] =', dd['key2'])
```

8.11.4. OrderedDict

使用 dict 时，Key 是无序的。在对 dict 做迭代时，我们无法确定 Key 的顺序。
如果要保持 Key 的顺序，可以用 OrderedDict，它的 Key 会按照插入的顺序排列，不是 Key 本身排序：

```
from collections import OrderedDict
od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
print(od)
# OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

OrderedDict 可以实现一个 FIFO（先进先出）的 dict，当容量超出限制时，先删除最早添加的 Key：

```
from collections import OrderedDict
class LastUpdatedOrderedDict(OrderedDict):

    def __init__(self, capacity):
        super(LastUpdatedOrderedDict, self).__init__()
        self._capacity = capacity

    def __setitem__(self, key, value):
        containsKey = 1 if key in self else 0
        if len(self) - containsKey >= self._capacity:
            last = self.popitem(last=False)
            print('remove:', last)
        if containsKey:
            del self[key]
            print('set:', (key, value))
        else:
            print('add:', (key, value))
        OrderedDict.__setitem__(self, key, value)
```

8.11.5. ChainMap&命令行参数解析

`ChainMap` 可以把一组 `dict` 串起来并组成一个逻辑上的 `dict`。`ChainMap` 本身也是一个 dict，但是查找的时候，会按照顺序在内部的 dict 依次查找。

什么时候使用 `ChainMap` 最合适？举个例子：应用程序往往都需要传入参数，参数可以通过命令行传入，可以通过环境变量传入，还可以有默认参数。我们可以用 `ChainMap` 实现参数的优先级查找，即先查命令行参数，如果没有传入，再查环境变量，如果没有，就使用默认参数。

下面的代码演示了如何查找 `user` 和 `color` 这两个参数：

```
from collections import ChainMap
import os, argparse

# 构造缺省参数:
defaults = {
```

```
    'color': 'red',
    'user': 'guest'
}

# 构造命令行参数:
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = { k: v for k, v in vars(namespace).items() if v }

# 组合成ChainMap:
combined = ChainMap(command_line_args, os.environ, defaults)

# 打印参数:
print('color=%s' % combined['color'])
print('user=%s' % combined['user'])
```

8.11.6. Counter

Counter 是 dict 的一个子类，可作为简单的计数器，例如，统计字符出现的个数：

```
from collections import Counter

c = Counter()
for ch in 'programming':
    c[ch] += 1
print(c)
```

结果：

Counter({'r': 2, 'g': 2, 'm': 2, 'p': 1, 'o': 1, 'a': 1, 'i': 1, 'n': 1})

8.12. itertools

Python 的内建模块 itertools 提供了非常有用的用于操作迭代对象的函数。

首先，我们看看 itertools 提供的几个“无限”迭代器：

8.12.1. count()

数值无限递增：

```
import itertools

natuals = itertools.count(1)
for n in natuals:
    print(n)
    if n >= 100:
        break
```

cycle()会把传入的一个序列无限重复下去：

```
cs = itertools.cycle('ABC')
t = 10
```

```
for c in cs:
    print(c)
    t = t - 1
    if t == 0:
        break
```

8.12.2. repeat()

`repeat()`负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

```
cs = itertools.repeat('A', 3)
for c in cs:
    print(c)
```

无限序列虽然可以无限迭代下去，但是通常我们会通过`takewhile()`等函数根据条件判断来截取出一个有限的序列：

```
natuals = itertools.count(1)
ns = itertools.takewhile(lambda x: x <= 10, natuals)
print(list(ns))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

8.12.3. chain()

`chain()`可以把一组迭代对象串联起来，形成一个更大的迭代器：

```
for c in itertools.chain('ABC', 'XYZ'):
    print(c)
```

8.12.4. groupby()

`groupby()`把迭代器中相邻的重复元素挑出来放在一起：

```
for key, group in itertools.groupby('AAABBBCCAAA'):
    print(key, list(group))
```

```
A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']
```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的 key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的 key：

```
for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.upper()):
    print(key, list(group))
```

```
A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']
```

计算圆周率:

$$\because \sum_{n=1}^{\infty}\frac{1}{n^2}=\frac{\pi^2}{6}(\text{由Euler发现})$$

$$\therefore \pi=\sqrt{6\cdot\sum_{n=1}^{\infty}\frac{1}{n^2}}$$

$$=\sqrt{6\cdot(1+\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}+\frac{1}{5^2}+\ldots)}$$

```
def pi1(N):
    ' 计算pi 的值 '
    sum = 0
    for i in itertools.count(1):
        if (i > N): break
        sum += 1 / i ** 2
    return math.sqrt(6 * sum)
```

$$\because \arctan x=\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{2n+1}(-1\le x\le 1)$$

$$\therefore \pi=4\cdot\arctan 1=4\cdot\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}$$

$$=4\cdot(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}+\ldots)$$

```
def pi2(N):
    ' 计算pi 的值 '
    sum=0
    i=0
    for count in itertools.count(1):
```

```
        i+=1
        if(i>N):break
        if(i%2==1):sum+=4/count
        else:sum-=4/count
    return sum
```

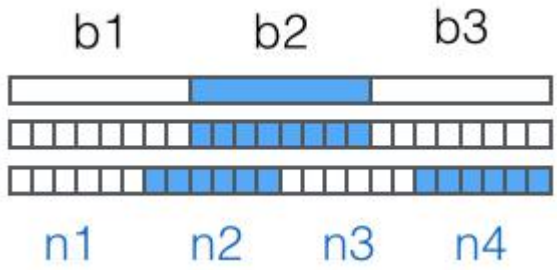
8.13. base64 编码

```
import base64

s = base64.b64encode('在 Python 中使用 BASE 64 编码'.encode('utf-8'))
print(s)
d = base64.b64decode(s).decode('utf-8')
print(d)

s = base64.urlsafe_b64encode('在 Python 中使用 BASE 64 编码'.encode('utf-8'))
print(s)
d = base64.urlsafe_b64decode(s).decode('utf-8')
print(d)
```

Base64 是一种任意二进制到文本字符串的编码方法，常用于在 URL、Cookie、网页中传输少量二进制数据。
编码原理：
首先，准备一个包含 64 个字符的数组：

```
['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']
```

然后，对二进制数据进行处理，每 3 个字节一组，一共是 3x8=24bit，划为 4 组，每组正好 6 个 bit：



这样我们得到 4 个数字作为索引，然后查表，获得相应的 4 个字符，就是编码后的字符串。
所以，Base64 编码会把 3 字节的二进制数据编码为 4 字节的文本数据，长度增加 33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。
如果要编码的二进制数据不是 3 的倍数，最后会剩下 1 个或 2 个字节，Base64 用\x00 字节在末尾补足后再编码，多余\x00 会被编码成=号，Base64 编码的最终结果长度将是 4 的倍数。
Python 内置的 base64 可以直接进行 base64 的编解码：

```
import base64

s = base64.b64encode('在 Python 中使用 BASE 64 编码'.encode('utf-8'))
print(s)# b'5ZyoUHl0aG9u5Lit5L2/55SoQkFTRSA2NOe8lueggQ=='
```

```
d = base64.b64decode(s).decode('utf-8')
print(d)
```

由于标准的 Base64 编码后可能出现字符+和/，在 URL 中就不能直接作为参数，所以又有一种"url safe"的 base64 编码，其实就是把字符+和/分别变成-和_：

```
s = base64.urlsafe_b64encode('在 Python 中使用 BASE 64 编码'.encode('utf-8'))
print(s)#b'5ZyoUHl0aG9u5Lit5L2_55SoQkFTRSA2NOe8lueggQ=='
d = base64.urlsafe_b64decode(s).decode('utf-8')
print(d)
```

还可以自己定义 64 个字符的排列顺序，这样就可以自定义 Base64 编码，不过，通常情况下完全没有必要。

由于=字符也可能出现在 Base64 编码中，但=用在 URL、Cookie 里面会造成歧义，所以，很多 Base64 编码后会把=去掉：

```
# 标准Base64:'abcd' -> 'YWJjZA=='
# 自动去掉=:'abcd' -> 'YWJjZA'
```

因为 Base64 是把 3 个字节变为 4 个字节，所以，Base64 编码的长度永远是 4 的倍数，因此，解码的时候只需要加上=把 Base64 字符串的长度变为 4 的倍数，就可以正常解码了。

示例：

```
def safe_base64_decode(s):
    s += b"=" * (4-len(s) % 4)
    return base64.urlsafe_b64decode(s)
s = base64.urlsafe_b64encode('在 Python 中使用 BASE 64 编码'.encode('utf-8'))
s = s.decode('utf-8').rstrip("=")
print(s)
d = safe_base64_decode(s.encode("utf-8")).decode('utf-8')
print(d)
```

8.14. struct 二进制数据的转换

Python 提供了一个 struct 模块来解决 bytes 和其他二进制数据类型的转换。

struct 的 pack 函数把任意数据类型变成 bytes。

```
import struct
print(struct.pack('>I', 10240099))
```

```
b'\x00\x9c@c'
```

pack 的第一个参数是处理指令：

>表示字节顺序是 big-endian，也就是网络序，

`I` 表示 4 字节无符号整数，

`H`：2 字节无符号整数。

后面的参数字节个数要和处理指令一致。

unpack 把 bytes 变成相应的数据类型：

```
>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')
```

```
(4042322160, 32896)
```

`struct`模块定义的数据类型可以参考 Python 官方文档：

https://docs.python.org/zh-cn/3/library/struct.html#format-characters

格式	C 类型	Python 类型	标准大小	注释
x	填充字节	无		
c	char	长度为 1 的字节串	1	
b	signed char	整数	1	(1), (2)
B	unsigned char	整数	1	-2
?	_Bool	bool	1	-1
h	short	整数	2	-2
H	unsigned short	整数	2	-2
i	int	整数	4	-2
I	unsigned int	整数	4	-2
l	long	整数	4	-2
L	unsigned long	整数	4	-2
q	long long	整数	8	-2
Q	unsigned long long	整数	8	-2
n	ssize_t	整数		-3
N	size_t	整数		-3
e	-6	浮点数	2	-4
f	float	浮点数	4	-4
d	double	浮点数	8	-4
s	char[]	字节串		
p	char[]	字节串		
P	void *	整数		-5

bmp 位图分析示例

Windows 的位图文件（.bmp）是一种非常简单的文件格式，采用小端方式存储数据。

文件头的结构按顺序如下：

- 两个字节：'BM'表示 Windows 位图，'BA'表示 OS/2 位图；
- 一个 4 字节整数：表示位图大小；
- 一个 4 字节整数：保留位，始终为 0；
- 一个 4 字节整数：实际图像的偏移量；
- 一个 4 字节整数：Header 的字节数；
- 一个 4 字节整数：图像宽度；
- 一个 4 字节整数：图像高度；
- 一个 2 字节整数：始终为 1；
- 一个 2 字节整数：颜色数。

读入前 30 个字节来分析：

```
import struct

bmp_header =
b'\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x
00\x00\x01\x00\x18\x00'

print(struct.unpack('<ccIIIIIIHH', bmp_header))
```

(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)

结果显示，b'B'、b'M'说明是 Windows 位图，位图大小为 640x360，颜色数为 24。

检查文件是否是位图文件，如果是，打印出图片大小和颜色数：

示例：

```
bmp_data = base64.b64decode(

'Qk1oAgAAAAAAADYAAAAoAAAAHAAAAAoAAAABABAAAAAAAADICAAASCwAAEgsAAAAAAAAAAAAA/3//f/9//3//f/9//3//f/9//3/
/f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9/AHwAfAB8AHwAfAB8AHwAfP9//3//fwB8AHwAfAB8/3//f/9
/AHwAfAB8AHz/f/9//3//f/9//38AfAB8AHwAfAB8AHwAfAB8AHz/f/9//38AfAB8/3//f/9//3//fwB8AHz/f/9//3//f/9//3/
/f/9/AHwAfP9//3//f/9/AHwAfP9//3//fwB8AHz/f/9//3//f/9//38AfAB8AHwAfAB8AHwAfP9
//3//f/9/AHwAfP9//3//f/9//38AfAB8/3//f/9//3//fwB8AHwAfAB8AHwAfAB8/3//f/9//38AfAB8/3//f/9//3/
/fwB8AHz/f/9//3//f/9/AHwAfP9//3//f/9/AHwAfP9//3//fwB8AHz/f/9/AHz/f/9/AHwAfP9//3//f/9/AHwAfAB8AHwAfAB8/3/
/f/9/AHwAfAB8AHz/fwB8AHwAfAB8AHwAfAB8AHz/f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9
//3//f/9//3//f/9//38AAA==')


def bmp_info(data):
    unpack = struct.unpack('<ccIIIIIIHH', data[:30])
    return {
        'width': unpack[6],
        'height': unpack[7],
        'color': unpack[9]
    }
bi = bmp_info(bmp_data)
assert bi['width'] == 28
assert bi['height'] == 10
assert bi['color'] == 16
print('ok')
```

8.15. hashlib 摘要算法

hashlib 提供了常见的摘要算法，如 MD5，SHA1 等等。

摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用 16 进制的字符串表示）。

摘要函数是一个单向函数通过摘要函数 `f()`对任意长度的数据 `data` 计算出固定长度的摘要 `digest`，计算 `f(data)` 很容易，但通过 `digest` 反推 `data` 却非常困难。而且，对原始数据做一个 bit 的修改，都会导致计算出的摘要完全不同。摘要算法可用于发现原始数据是否被人篡改过。

8.15.1. MD5&SHA1

计算出一个字符串的 MD5 值：

```
import hashlib

md5 = hashlib.md5()
md5.update(b'how to use md5 in python hashlib?')
print(md5.hexdigest())
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用 update()，最后计算的结果是一样的：

```
import hashlib

md5 = hashlib.md5()
md5.update(b'how to use md5 in ')
md5.update(b'python hashlib?')
print(md5.hexdigest())
```

MD5 是最常见的摘要算法，速度很快，生成结果是固定的 128 bit 字节，通常用一个 32 位的 16 进制字符串表示。

另一种常见的摘要算法是 SHA1，调用 SHA1 和调用 MD5 完全类似：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update(b'how to use sha1 in ')
sha1.update(b'python hashlib?')
print(sha1.hexdigest())
```

SHA1 的结果是 160 bit 字节，通常用一个 40 位的 16 进制字符串表示。
比 SHA1 更安全的算法是 SHA256 和 SHA512，不过越安全的算法不仅越慢，而且摘要长度更长。

因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中，两个不同的数据通过某个摘要算法有可能能得到相同的摘要。这种情况称为碰撞，但是概率非常非常低。

8.15.2. hmac

对于用户密码可以保存对应的 MD5 值，用保存在数据库中的 password_md5 对比计算 md5(password)的结果，如果一致，用户输入的口令就是正确的。存储 MD5 的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。
采用 MD5 存储口令也不一定安全。很多用户喜欢用 123456，888888，password 这些简单的口令，于是，黑客可以事先计算出这些常用口令的 MD5 值，得到一个反推表：

```
'e10adc3949ba59abbe56e057f20f883e': '123456'

'21218cca77804d2ba1922c33e0151105': '888888'

'5f4dcc3b5aa765d61d8327deb882cf99': 'password'
```

这样，无需破解，只需要对比数据库的 MD5，黑客就获得了使用常用口令的用户账号。
对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？
由于常用口令的 MD5 值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的 MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

```
def calc_md5(password):

    return get_md5(password + 'the-Salt')
```

经过 Salt 处理的 MD5 口令，只要 Salt 不被黑客知道，即使用户输入简单口令，也很难通过 MD5 反推明文口令。
但是如果有两个用户都使用了相同的简单口令比如 123456，在数据库中，将存储两条相同的 MD5 值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的 MD5 呢？
如果假定用户无法修改登录名，就可以通过把登录名作为 Salt 的一部分来计算 MD5，从而实现相同口令的用户也存储不同的 MD5。

如果 salt 是我们自己随机生成的，通常我们计算 MD5 时采用 md5(message + salt)。但实际上，把 salt 看做一个“口令”，加 salt 的哈希就是：计算一段 message 的哈希时，根据不同口令计算出不同的哈希。要验证哈希值，必须同时提供正确的口令。
这实际上就是 Hmac 算法：Keyed-Hashing for Message Authentication。它通过一个标准算法，在计算哈希的过程中，把 key 混入计算过程中。
和我们自定义的加 salt 算法不同，Hmac 算法针对所有哈希算法都通用，无论是 MD5 还是 SHA-1。采用 Hmac 替代我们自己的 salt 算法，可以使程序算法更标准化，也更安全。
Python 自带的 hmac 模块实现了标准的 Hmac 算法。我们来看看如何使用 hmac 实现带 key 的哈希。
我们首先需要准备待计算的原始消息 message，随机 key，哈希算法，这里采用 MD5，使用 hmac 的代码如下：

```
import hmac
message = b'Hello, world!'
key = b'secret'
h = hmac.new(key, message, digestmod='MD5')
# 如果消息很长，可以多次调用h.update(msg)
print(h.hexdigest())
# fa4ee7d173f2d97ee79022d1a7355bcf
```

8.16. contextlib

在 Python 中，读写文件资源必须在使用完毕后正确关闭它们。正确关闭文件资源的一个方法是使用 try...finally,写 try...finally 非常繁琐。Python 的 with 语句允许我们非常方便地使用资源，而不必担心资源没有关闭
任何对象，只要正确实现了上下文管理，就可以用于 with 语句。
实现上下文管理是通过__enter__和__exit__这两个方法实现的。
例如，下面的 class 实现了这两个方法：

```
class Query(object):

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

    def __enter__(self):
        print('Begin')
        return self
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type:
            print('Error')
        else:
            print('End')

    def query(self):
        print('Query info about %s...' % self.name)
```

这样我们就可以把自己写的资源对象用于 with 语句：

```
with Query('Bob') as q:
    q.query()
```

8.16.1. @contextmanager

编写__enter__和__exit__仍然很繁琐，因此 Python 的标准库 contextlib 提供了更简单的写法，上面的代码可以改写如下：

```
from contextlib import contextmanager

class Query(object):

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

    def query(self):
        print('Query info about %s...' % self.name)

@contextmanager
def create_query(name):
    print('Begin')
    q = Query(name)
    yield q
    print('End')
```

`@contextmanager` 这个 decorator 接受一个 generator，用 `yield` 语句把 `with ... as var` 把变量输出出去，然后，`with` 语句就可以正常地工作了：

```
with create_query('Bob') as q:
    q.query()
```

很多时候，我们希望在某段代码执行前后自动执行特定代码，也可以用`@contextmanager` 实现。例如：

```
@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)
with tag("h1"):
```

```
    print("hello")
    print("world")
```

代码的执行顺序是：

1. with 语句首先执行 yield 之前的语句，因此打印出<h1>；
2. yield 调用会执行 with 语句内部的所有语句，因此打印出 hello 和 world；
3. 最后执行 yield 之后的语句，打印出</h1>。
4. 因此，@contextmanager 让我们通过编写 generator 来简化上下文管理。

8.16.2. @closing

如果一个对象没有实现上下文，我们就不能把它用于 with 语句。这个时候，可以用 closing()来把该对象变为上下文对象。例如，用 with 语句使用 urlopen()：

```
from contextlib import closing
from urllib.request import urlopen
with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

closing 也是一个经过@contextmanager 装饰的 generator，这个 generator 编写起来其实非常简单：

```
@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

它的作用就是把有 close()方法的对象变为上下文对象，并支持 with 语句。

8.17. urllib

urllib 提供了一系列用于操作 URL 的功能。

8.17.1. Get

urllib 的 `request` 模块可以非常方便地抓取 URL 内容，也就是发送一个 GET 请求到指定的页面，然后返回 HTTP 的响应：

```
from urllib import request
with request.urlopen('https://suggest.taobao.com/sug?code=utf-8&q=python') as f:
    data = f.read()
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', data.decode('utf-8'))
```

可以看到 HTTP 响应的头和 JSON 数据：

```
Status: 200 OK
Date: Sat, 05 Oct 2019 01:28:54 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
```

Connection: close
Vary: Accept-Encoding
Server: Tengine/Aserver
Strict-Transport-Security: max-age=31536000
Timing-Allow-Origin: *
EagleEye-TraceId: 0b802cec15702389349248065ec6d8
Data:
{"result":[["python 编程从入门","4480.345510835913"],["python 语言程序设计","4975.16191904048"],["python 基础教程","6708.292708333333"],["python 入门","13308.612213740458"],["python 二手书","119.95141700404858"],["python 二级","913.5652173913044"],["python 编程从入门到实战","1046.361344537815"],["python 语言程序设计基础","2095.4222222222224"],["python 程序设计基础","4021.3518518518517"],["python 学习手册","1012.0629370629371"]]}

如果我们要想模拟浏览器发送 GET 请求，就需要使用 `Request` 对象，通过往 `Request` 对象添加 HTTP 头，我们就可以把请求伪装成浏览器。例如，模拟 iPhone 6 去请求豆瓣首页：

```
from urllib import request

req = request.Request('http://www.douban.com/')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')
with request.urlopen(req) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

这样豆瓣会返回适合 iPhone 的移动版网页.

8.17.2. Post

如果要以 POST 发送一个请求，只需要把参数 `data` 以 bytes 形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照 weibo.cn 的登录页的格式以 `username=xxx&password=xxx` 的编码传入：

```
from urllib import request, parse

print('Login to post page...')
email = input('Email: ')
passwd = input('Password: ')
login_data = parse.urlencode([
    ('username', email),
    ('password', passwd)
])

req = request.Request(' http://127.0.0.1:5000/signin')
req.add_header('User-Agent', 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36')
req.add_header('Referer',
```

```
'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')
with request.urlopen(req, data=login_data.encode('utf-8')) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

8.17.3. Handler

如果还需要更复杂的控制，比如通过一个 Proxy 去访问网站，我们需要利用 ProxyHandler 来处理，示例代码如下：

```
import urllib

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)
with opener.open('http://www.example.com/login.html') as f:
    pass
```

8.18. requests

Python 内置的 urllib 模块用于访问网络资源。但是，它用起来比较麻烦，而且，缺少很多实用的高级功能。
更好的方案是使用 requests。它是一个 Python 第三方库，处理 URL 资源特别方便。

如果安装了 Anaconda，requests 就已经可用了。否则，需要在命令行下通过 pip 安装：

```
pip install requests
```

8.18.1. GET 访问

```
import requests
r = requests.get('https://www.douban.com/') # 豆瓣首页

print(r.text)
```

传入一个 dict 作为 params 参数：

```
r = requests.get('https://www.douban.com/search', params={'q': 'python', 'cat': '1001'})
print(r.url) # 实际请求的URL
# https://www.douban.com/search?q=python&cat=1001
```

requests 自动检测编码，可以使用 encoding 属性查看：

```
print(r.encoding)
# utf-8
```

用 content 属性获得 bytes 二进制内容：

```
print(r.content)
```

对于特定类型的响应，例如 JSON，可以直接获取：

```
r = requests.get("https://www.tianqiapi.com/api/?version=v1&city=深圳
&appid=13639281&appsecret=yCJK9RVk")
print(r.json())
```

传入一个 dict 作为 headers 参数：

```
r = requests.get('https://www.douban.com/', headers={'User-Agent': 'Mozilla/5.0 (iPhone; CPU iPhone OS
11_0 like Mac OS X) AppleWebKit'})
print(r.text)
```

8.18.2. POST 访问

要发送 POST 请求，只需要把 get()方法变成 post()，然后传入 data 参数作为 POST 请求的数据：

```
r = requests.post('https://accounts.douban.com/login', data={'form_email': 'abc@example.com',
'form_password': '123456'})
```

requests 默认使用 `application/x-www-form-urlencoded` 对 POST 数据编码。如果要传递 JSON 数据，可以直接传入 json 参数：

```
params = {'key': 'value'}
r = requests.post(url, json=params) # 内部自动序列化为 JSON
```

类似的，上传文件需要更复杂的编码格式，但是 requests 把它简化成 files 参数：

```
upload_files = {'file': open('report.xls', 'rb')}
r = requests.post(url, files=upload_files)
```

在读取文件时，**务必使用`'rb'`即二进制模式读取，这样获取的 `bytes` 长度才是文件的长度**。

把 `post()` 方法替换为 `put()`，`delete()` 等，就可以以 PUT 或 DELETE 方式请求资源。

除了能轻松获取响应内容外，requests 对获取 HTTP 响应的其他信息也非常简单。例如，获取响应头：

```
r.headers
r.headers['Content-Type']
```

获取指定的 Cookie：

```
r.cookies['ts']
```

传入 `cookies` 参数：

```
cs = {'token': '12345', 'status': 'working'}
r = requests.get(url, cookies=cs)
```

指定 timeout 超时参数：

```
r = requests.get(url, timeout=2.5) # 2.5 秒后超时
```

8.19. 使用 chardet 推测编码

如果安装了 Anaconda，chardet 就已经可用了。否则，需要在命令行下通过 pip 安装：
pip install chardet

```
import chardet

print(chardet.detect(b'Hello, world!'))
print(chardet.detect('离离原上草，一岁一枯荣'.encode('gbk')))
print(chardet.detect('离离原上草，一岁一枯荣'.encode('utf-8')))
print(chardet.detect('最新の主要ニュース'.encode('euc-jp')))
```

运行结果：

```
{'encoding': 'ascii', 'confidence': 1.0, 'language': ''}
{'encoding': 'GB2312', 'confidence': 0.7407407407407407, 'language': 'Chinese'}
{'encoding': 'utf-8', 'confidence': 0.99, 'language': ''}
{'encoding': 'EUC-JP', 'confidence': 0.99, 'language': 'Japanese'}
```

'encoding'表示检测结果；confidence 表示检测的概率。
注意 GBK 包含 GB2312 编码。

智能解码示例：

```
import chardet, requests
content = requests.get("http://www.baidu.com").content
print(content.decode(chardet.detect(content).get('encoding')))
```

8.20. Pillow

PIL：Python Imaging Library，已经是 Python 平台事实上的图像处理标准库了。PIL 功能非常强大，但 API 却非常简单易用。
由于 PIL 仅支持到 Python 2.7，加上年久失修，于是一群志愿者在 PIL 的基础上创建了兼容的版本，名字叫 Pillow，支持最新 Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用 Pillow。
如果安装了 Anaconda，Pillow 就已经可用了。否则，需要在命令行下通过 pip 安装：
pip install pillow
详细内容：

https://pillow.readthedocs.org/

8.20.1. 图像缩放操作

```
from PIL import Image

# 打开一个jpg图像文件，注意是当前路径:
im = Image.open('test.jpg')
# 获得图像尺寸:
w, h = im.size
print('Original image size: %sx%s' % (w, h))
```

```
# 缩放到 50%:
im.thumbnail((w//2, h//2))
print('Resize image to: %sx%s' % (w//2, h//2))
# 把缩放后的图像用 jpeg 格式保存:
im.save('thumbnail.jpg', 'jpeg')
```

8.20.2. 模糊效果

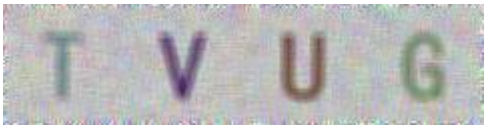
```
from PIL import Image, ImageFilter

# 打开一个 jpg 图像文件，注意是当前路径:
im = Image.open('test.jpg')
# 应用模糊滤镜:
im2 = im.filter(ImageFilter.BLUR)
im2.save('blur.jpg', 'jpeg')
```

8.20.3. ImageDraw 生成字母验证码图片

```
from PIL import Image, ImageDraw, ImageFont, ImageFilter
import random

# 随机字母:
def rndChar():
    return chr(random.randint(ord('A'), ord("Z")))
# 随机颜色 1:
def rndBackgroundColor():
    return (random.randint(128, 255), random.randint(128, 255), random.randint(128, 255))
# 随机颜色 2:
def rndFontColor():
    return (random.randint(32, 127), random.randint(32, 127), random.randint(32, 127))


# 240 x 60:
width = 60 * 4
height = 60
image = Image.new('RGB', (width, height), (255, 255, 255))

# 创建 Draw 对象:
draw = ImageDraw.Draw(image)
# 填充背景:
for x in range(width):
    for y in range(height):
        draw.point((x, y), fill=rndBackgroundColor())

# 创建 Font 对象，并输出文字:
font = ImageFont.truetype('simkai.ttf', 48)
result = []
for t in range(4):
    ch = rndChar()
```

```
    result.append(ch)
    draw.text((60 * t + 18, 6), ch, font=font, fill=rndFontColor())

print("验证码：", "".join(result))

# 模糊:
image = image.filter(ImageFilter.BLUR)
image.save('code.jpg', 'jpeg')
```

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片如下：



8.21. psutil

psutil = process and system utilities，它不仅可以通过一两行代码实现系统监控，还可以跨平台使用。
需要在命令行下通过 pip 安装：

```
pip install psutil
```

参见：

https://github.com/giampaolo/psutil

8.21.1. 获取 CPU 信息

```
import psutil

print(psutil.cpu_count()) # CPU 逻辑数量 4
print(psutil.cpu_count(logical=False)) # CPU 物理核心 2
# 2 说明是双核超线程, 4 则是 4 核非超线程
```

8.21.2. 统计 CPU 的用户 / 系统 / 空闲时间

```
print(psutil.cpu_times())
#scputimes(user=6326.5737547, system=2843.976230500004, idle=56593.7151779,
interrupt=180.05635420000002, dpc=206.4049231)
```

再实现类似 top 命令的 CPU 使用率，每秒刷新一次，累计 10 次：

```
for x in range(10):
    print(psutil.cpu_percent(interval=1, percpu=True))
```

8.21.3. 获取内存信息

使用 psutil 获取物理内存和交换内存信息，分别使用：

```
print(psutil.virtual_memory())
#svmem(total=8375676928, available=4138205184, percent=50.6, used=4237471744, free=4138205184)
print(psutil.swap_memory())
#sswap(total=16749408256, used=5296283648, free=11453124608, percent=31.6, sin=0, sout=0)
```

返回的是字节为单位的整数。

8.21.4. 获取磁盘信息

可以通过 psutil 获取磁盘分区、磁盘使用率和磁盘 IO 信息：

```
print(psutil.disk_partitions()) # 磁盘分区信息
# [sdiskpart(device='C:\\', mountpoint='C:\\', fstype='NTFS', opts='rw,fixed'),
# sdiskpart(device='D:\\', mountpoint='D:\\', fstype='NTFS', opts='rw,fixed'),
# sdiskpart(device='G:\\', mountpoint='G:\\', fstype='', opts='removable'),
# sdiskpart(device='I:\\', mountpoint='I:\\', fstype='', opts='cdrom')]
print(psutil.disk_usage('D:\\')) # 磁盘使用情况
# sdiskusage(total=93417631744, used=73671778304, free=19745853440, percent=78.9)
print(psutil.disk_io_counters()) # 磁盘 IO
# sdiskio(read_count=204367, write_count=196261, read_bytes=5218053632, write_bytes=7368294912,
read_time=195, write_time=94)
```

8.21.5. 获取网络信息

psutil 可以获取网络接口和网络连接信息：

```
print(psutil.net_io_counters()) # 获取网络读写字节／包的个数
# ssnetio(bytes_sent=353226381, bytes_recv=124974975, packets_sent=239876, packets_recv=287971,
errin=0, errout=0, dropin=0, dropout=0)
print(psutil.net_if_addrs()) # 获取网络接口信息
print(psutil.net_if_stats()) # 获取网络接口状态
```

要获取当前网络连接信息，使用 net_connections()：

```
print(psutil.net_connections()) # 获取当前网络连接信息
```

注意：获取网络连接信息需要 root 权限。

8.21.6. 获取进程信息

通过 psutil 可以获取到所有进程的详细信息：

```
>>> psutil.pids() # 所有进程 ID
>>> p = psutil.Process(3776) # 获取指定进程 ID=3776，其实就是当前 Python 交互环境
>>> p.name() # 进程名称
>>> p.exe() # 进程 exe 路径
>>> p.cwd() # 进程工作目录
>>> p.cmdline() # 进程启动的命令行
>>> p.ppid() # 父进程 ID
>>> p.parent() # 父进程
<psutil.Process(pid=3765, name='bash') at 4503144040>
>>> p.children() # 子进程列表
```

```
>>> p.status() # 进程状态
>>> p.username() # 进程用户名
>>> p.create_time() # 进程创建时间
>>> p.terminal() # 进程终端
>>> p.cpu_times() # 进程使用的 CPU 时间
>>> p.memory_info() # 进程使用的内存
>>> p.open_files() # 进程打开的文件
>>> p.connections() # 进程相关网络连接
>>> p.num_threads() # 进程的线程数量
>>> p.threads() # 所有线程信息
>>> p.environ() # 进程环境变量
>>> p.terminate() # 结束进程
```

8.21.7. 模拟 ps 命令的效果

psutil 提供了 test()函数可以模拟出 ps 命令的效果：

```
psutil.test()
```

8.22. 海龟绘图 Turtle

在 1966 年，Seymour Papert 和 Wally Feurzig 发明了一种专门给儿童学习编程的语言——LOGO 语言，它的特色就是通过编程指挥一个小海龟（turtle）在屏幕上绘图。

海龟绘图（Turtle Graphics）后来被移植到各种高级语言中，Python 内置了 turtle 库，基本上 100%复制了原始的 Turtle Graphics 的所有功能。

我们来看一个指挥小海龟绘制一个长方形的简单代码：

```
# 导入 turtle 包的所有内容:
from turtle import *

# 设置笔刷宽度:
width(4)
# 前进:
forward(200)
# 右转 90 度:
right(90)
# 笔刷颜色:
pencolor('red')
forward(100)
right(90)
pencolor('green')
```

```
forward(200)
right(90)
pencolor('blue')
forward(100)
right(90)

# 调用 done()使得窗口等待被关闭，否则将立刻关闭窗口:

done()
```

在命令行运行上述代码，会自动弹出一个绘图窗口，然后绘制出一个长方形.

调用 `width()`函数可以设置笔刷宽度，调用 `pencolor()`函数可以设置颜色。

更多操作请参考:https://docs.python.org/zh-cn/3.7/library/turtle.html

通过循环绘制 5 个五角星：

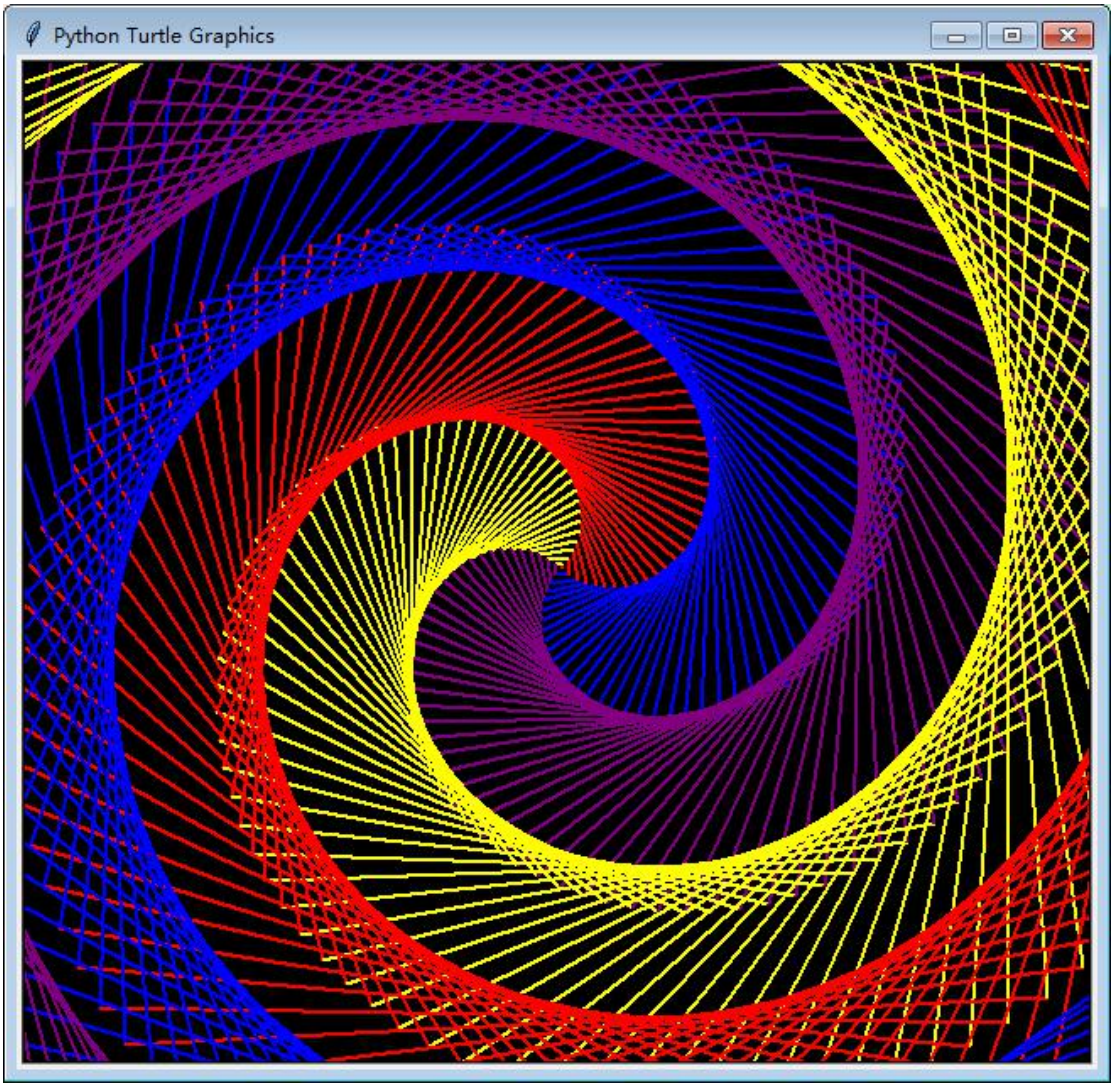
```
from turtle import *

def drawStar(x, y):
    pu()
    goto(x, y)
    pd()
    # set heading: 0
    seth(0)
    for i in range(5):
        fd(40)
        rt(144)

for x in range(0, 250, 50):
    drawStar(x, 0)

done()
```

程序执行效果如下：

彩色螺旋线：

```
import turtle
turtle.pensize(2)
turtle.bgcolor("black")
colors = ["red","yellow",'purple','blue']
turtle.tracer(False)
for x in range (400):
    turtle.forward(2*x)
    turtle.color(colors[x % 4])
    turtle.left(91)
```

```
turtle.tracer(True)
turtle.done()
```

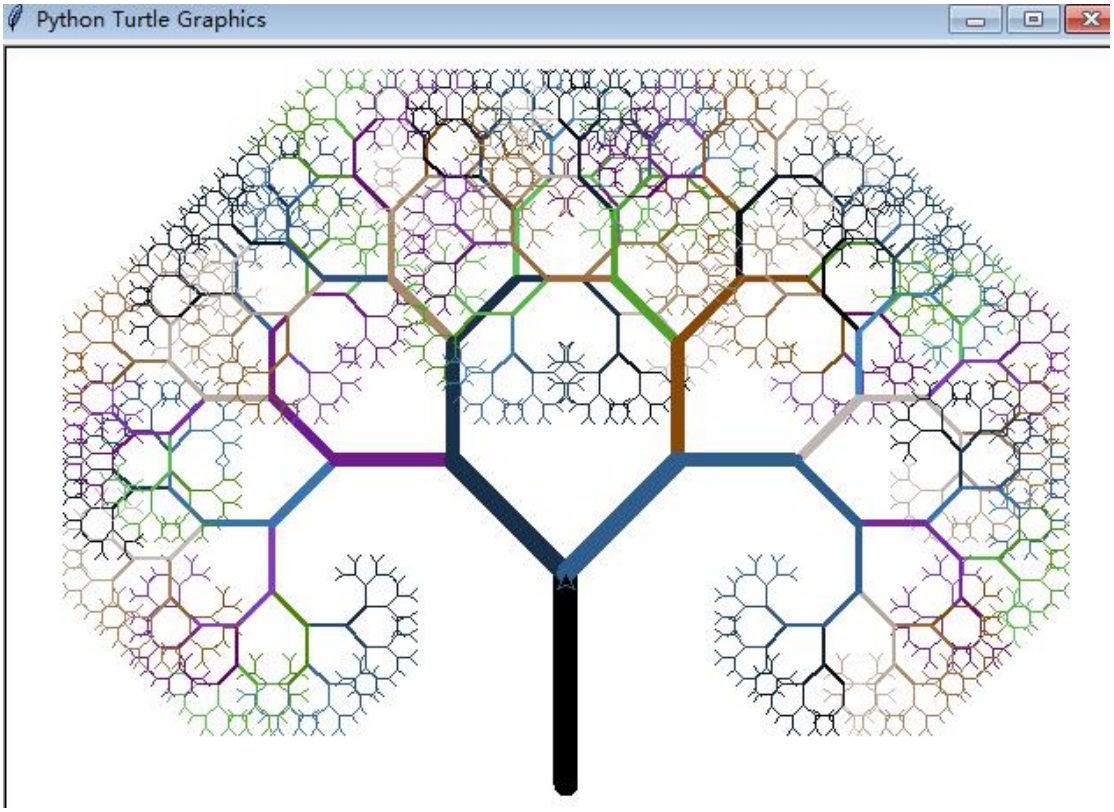


使用递归，可以绘制出非常复杂的图形。例如，下面的代码可以绘制一棵分型树：

```
from turtle import *

# 设置色彩模式是 RGB:

colormode(255)

lt(90)

lv = 14
l = 120
s = 45

width(lv)

r = 0
g = 0
b = 0
pencolor(r, g, b)

penup()
bk(l)
```

```
pendown()
fd(l)

def draw_tree(l, level):
    global r, g, b
    # save the current pen width
    w = width()

    # narrow the pen width
    width(w * 3.0 / 4.0)
    # set color:
    r = r + 1
    g = g + 2
    b = b + 3
    pencolor(r % 200, g % 200, b % 200)

    l = 3.0 / 4.0 * l

    lt(s)
    fd(l)

    if level < lv:
        draw_tree(l, level + 1)
    bk(l)
    rt(2 * s)
    fd(l)

    if level < lv:
        draw_tree(l, level + 1)
    bk(l)
    lt(s)

    # restore the previous pen width
    width(w)

speed("fastest")

draw_tree(l, 4)

done()
```

执行上述程序需要花费一定的时间，最后的效果如下：



9. Python3 正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。

re 模块，它提供 Perl 风格的正则表达式模式。

9.1. Re 模块

9.1.1. re.match&re.search 函数

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

re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回 none。

函数语法：

```
re.match(pattern, string, flags=0)
```

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

函数语法：

```
re.search(pattern, string, flags=0)
```

函数参数说明：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功 re.match 方法返回一个匹配的对象，否则返回 None：

```
import re

print(re.match('www', 'www. taobao.com').span())  # 在起始位置匹配

print(re.match('com', 'www. taobao.com'))         # 不在起始位置匹配

print(re.search('www', 'www. taobao.com').span())  # 在起始位置匹配

print(re.search('com', 'www. taobao.com').span())         # 不在起始位置匹配
```

以上实例运行输出结果为：

```
(0, 3)
None
(0, 3)
(11, 14)
```

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

匹配对象方法	描述
group(num=0)	匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。
groups()	返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。

```
import re

line = "Cats are smarter than dogs"

# .* 表示任意匹配除换行符（\n、\r）之外的任何单个或多个字符

matchObj = re.match(r'(.*) are (.*?) .*', line, re.M | re.I)

if matchObj:
    print("matchObj.group() : ", matchObj.group())
    print("matchObj.group(1) : ", matchObj.group(1))
    print("matchObj.group(2) : ", matchObj.group(2))
else:
    print("No match!!")
```

以上实例执行结果如下：

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

实例

```
import re

line = "Cats are smarter than dogs"

searchObj = re.search(r'(.*) are (.*?) .*', line, re.M | re.I)

if searchObj:
    print("searchObj.group() : ", searchObj.group())
    print("searchObj.group(1) : ", searchObj.group(1))
    print("searchObj.group(2) : ", searchObj.group(2))
else:
    print("Nothing found!!")
```

以上实例执行结果如下：

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

re.match&re.search 函数都将返回 re.MatchObject，方法列表：

- group() 返回被 RE 匹配的字符串。
- start() 返回匹配开始的位置
- end() 返回匹配结束的位置
- span() 返回一个元组包含匹配 (开始,结束) 的位置

9.1.2. re.sub 替换

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

语法：

```
re.sub(pattern, repl, string, count=0, flags=0)
```

参数：

- pattern ： 正则中的模式字符串。
- repl ： 替换的字符串，也可为一个函数。
- string ： 要被查找替换的原始字符串。
- count ： 模式匹配后替换的最大次数，默认 0 表示替换所有的匹配。

- flags：编译时用的匹配模式，数字形式。

实例

```
import re

phone = "2004-959-559 # 这是一个电话号码"


# 删除注释

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


# 移除非数字的内容

num = re.sub(r'\D', "", phone)
print("电话号码 : ", num)
```

以上实例执行结果如下：

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

9.1.3. repl 参数是一个函数

以下实例中将字符串中的匹配的数字乘于 2：

```
import re


# 将匹配的数字乘于 2

def double(matched):
    value = int(matched.group('value'))
    return str(value * 2)

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

执行输出结果为：

```
A46G8HFD1134
```

9.1.4. compile 编译正则表达式

re.compile() 返回 re.RegexObject 正则表达式（ Pattern ）对象。供 match() 和 search() 这两个函数使用。

语法格式为：

```
re.compile(pattern[, flags])
```

参数：

- pattern：一个字符串形式的正则表达式
- flags 可选，表示匹配模式，比如忽略大小写，多行模式等，具体参数为：
  - re.I 忽略大小写
  - re.L 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境
  - re.M 多行模式
  - re.S 即为'.'并且包括换行符在内的任意字符（'.'不包括换行符）
  - re.U 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库
  - re.X 为了增加可读性，忽略空格和'#'后面的注释

```
import re


pattern = re.compile(r'\d+')  # 用于匹配至少一个数字

m = pattern.match('one12twothree34four', 3, 10)  # 从'1'的位置开始匹配，正好匹配

print(m)  # 返回一个 Match 对象

print(m.group(0), m.start(0), m.end(0), m.span(0))
```

结果：

<re.Match object; span=(3, 5), match='12'>

12 3 5 (3, 5)

```
pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)  # re.I 表示忽略大小写

m = pattern.match('Hello World Wide Web')
print(m)
```

结果：

<re.Match object; span=(0, 11), match='Hello World'>

9.1.5. findall

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

注意： match 和 search 是匹配一次， findall 是匹配所有。

语法格式为：

```
re.findall(string[, pos[, endpos]])
```

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

查找字符串中的所有数字：

```
import re
```

```
pattern = re.compile(r'\d+')  # 查找数字

result1 = pattern.findall('  taobao 123 google 456')
result2 = pattern.findall('run88oob123google456', 0, 10)

print(result1)
print(result2)
```

输出结果：

```
['123', '456']
['88', '12']
```

9.1.6. re.finditer

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

```
re.finditer(pattern, string, flags=0)
```

```
import re

it = re.finditer(r"\d+", "12a32bc43jf3")
for match in it:
    print(match.group(),end=" ")
```

输出结果：

```
12 32 43 3
```

9.1.7. re.split

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

```
re.split(pattern, string[, maxsplit=0, flags=0])
```

```
import re

print(re.split('\W+', '  taobao,  taobao,  taobao.'))
# ['  taobao', '  taobao', '  taobao', '']
print(re.split('(\W+)', '  taobao,  taobao,  taobao.'))
# ['', '  ', '  taobao', ', ', '  taobao', ', ', '  taobao', '.', '']
print(re.split('\W+', '  taobao,  taobao,  taobao.', 1))
# ['', '  taobao,  taobao,  taobao.']
print(re.split('a+', 'hello world'))
# ['hello world']
```

9.2. 正则表达式修饰符 - 可选标志

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

- re.I 忽略大小写
- re.L 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境
- re.M 多行模式
- re.S 即为'.'并且包括换行符在内的任意字符（'.'不包括换行符）
- re.U 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库
- re.X 为了增加可读性，忽略空格和'#'后面的注释

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 $
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据 Unicode 字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

9.3. 正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。

多数字母和数字前加一个反斜杠时会拥有不同的含义。

标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。

反斜杠本身需要使用反斜杠转义。

由于正则表达式通常都包含反斜杠，所以你最好使用原始字符串来表示它们。模式元素(如 `r'\t'`，等价于 `\\t`)匹配相应的特殊字符。

9.3.1. 非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

字符	描述
\cx	匹配由 x 指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。

\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

9.3.2. 特殊字符

若要匹配这些特殊字符，必须首先使字符"转义"，即，将反斜杠字符\ 放在它们前面。下表列出了正则表达式中的特殊字符：

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。
[	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。

9.3.3. 限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有 * 或 + 或 ? 或 {n} 或 {n,} 或 {n,m} 共 6 种：

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "fooooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。

{n,m}	m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

9.3.4. 定位符

定位符使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。
定位符用来描述字符串或单词的边界，^ 和 $ 分别指字符串的开始与结束，\b 描述单词的前或后边界，\B 表示非单词边界。
正则表达式的定位符有：

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个单词边界，即字与空格间的位置。
\B	非单词边界匹配。

9.3.5. Python 正则表达式模式

下表列出了正则表达式模式语法中的特殊元素。如果你使用模式的同时提供了可选的标志参数，某些模式元素的含义会改变。

模式	描述
^	匹配字符串的开头
$	匹配字符串的末尾。
.	匹配任意字符，除了换行符，当 re.DOTALL 标记被指定时，则可以匹配包括换行符的任意字符。
[...]	用来表示一组字符,单独列出：[amk] 匹配 'a'，'m'或'k'
[^...]	不在[]中的字符：[^abc] 匹配除了 a,b,c 之外的字符。
re*	匹配 0 个或多个的表达式。
re+	匹配 1 个或多个的表达式。
re?	匹配 0 个或 1 个由前面的正则表达式定义的片段，非贪婪方式
re{ n}	匹配 n 个前面表达式。例如，"o{2}"不能匹配"Bob"中的"o"，但是能匹配"food"中的两个 o。
re{ n,}	精确匹配 n 个前面表达式。例如，"o{2,}"不能匹配"Bob"中的"o"，但能匹配"fooooood"中的所有 o。"o{1,}"等价于"o+"。"o{0,}"则等价于"o*"。
re{ n, m}	匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
a| b	匹配 a 或 b
(re)	匹配括号内的表达式，也表示一个组

(?imx)	正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。
(?-imx)	正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。
(?: re)	类似 (...), 但是不表示一个组
(?imx: re)	在括号中使用 i, m, 或 x 可选标志
(?-imx: re)	在括号中不使用 i, m, 或 x 可选标志
(?#...)	注释.
(?= re)	前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
(?! re)	前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功。
(?> re)	匹配的独立模式，省去回溯。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]
\d	匹配任意数字，等价于 [0-9]。
\D	匹配任意非数字
\A	匹配字符串开始
\Z	匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。
\z	匹配字符串结束
\G	匹配最后匹配完成的位置。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\n, \t, 等。	匹配一个换行符。匹配一个制表符, 等
\1...\9	匹配第 n 个分组的内容。
\10	匹配第 n 个分组的内容，如果它经匹配。否则指的是八进制字符码的表达式。

9.3.6. 通过?实现非贪婪匹配

正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 0：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()

('102300', '')
```

由于\d+采用贪婪匹配，直接把后面的 0 全部匹配了，结果 0*只能匹配空字符串了。

必须让\d+采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 0 匹配出来，加个?就可以让\d+采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()

('1023', '00')
```

提取 email：

```
def name_of_email(addr):
    return re.match(r'\W*([\w\s\.]+)', addr).group(1)

assert name_of_email('<Tom Paris> tom@voyager.org') == 'Tom Paris'
assert name_of_email('tom@voyager.org') == 'tom'
assert name_of_email('bill.gates@microsoft.com') == 'bill.gates'
```

采用非贪婪匹配实现：

```
def name_of_email(addr):
    return re.match(r'.*?([\w\s\.]+)', addr).group(1)
```

9.3.7. 前端正则表达式模式

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个 向后引用、或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。
$	匹配输入字符串的结束位置。如果设置了 RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。
.	匹配除换行符（\n、\r）之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用像"(.|\n)"的模式。

(pattern)	匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在 VBScript 中使用 SubMatches 集合，在 JScript 中则使用 $0…$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。
(?:pattern)	匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如， 'industr(?:y|ies) 就是一个比 'industry|industries' 更简略的表达式。
(?=pattern)	正向肯定预查（look ahead positive assert），在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，"Windows(?=95|98|NT|2000)"能匹配"Windows2000"中的"Windows"，但不能匹配"Windows3.1"中的"Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	正向否定预查(negative assert)，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如"Windows(?!95|98|NT|2000)"能匹配"Windows3.1"中的"Windows"，但不能匹配"Windows2000"中的"Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?<=pattern)	反向(look behind)肯定预查，与正向肯定预查类似，只是方向相反。例如，"(?<=95|98|NT|2000)Windows"能匹配"2000Windows"中的"Windows"，但不能匹配"3.1Windows"中的"Windows"。
(?<!pattern)	反向否定预查，与正向否定预查类似，只是方向相反。例如 "(?<!95|98|NT|2000)Windows"能匹配"3.1Windows"中的"Windows"，但不能匹配"2000Windows"中的"Windows"。
x|y	匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。
[xyz]	字符集合。匹配所包含的任意一个字符。例如，'[abc]' 可以匹配 "plain" 中的 'a'。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'、'l'、'i'、'n'。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\cx	匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 [^0-9]。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。

\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。
\w	匹配字母、数字、下划线。等价于'[A-Za-z0-9_]'。
\W	匹配非字母、数字、下划线。等价于 '[^A-Za-z0-9_]'。
\xn	匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。
\num	匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。
\n	标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。
\nm	标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。
\nml	如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。
\un	匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (?)。

示例：

正则表达式	描述
/\b([a-z]+) \1\b/gi	一个单词连续出现的位置。
/(\w+):\/\/([^/:]+)(:\d*)?([^#]*)/	将一个 URL 解析为协议、域、端口及相对路径。
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/	定位章节的位置。
/[-a-z]/	a 至 z 共 26 个字母再加一个-号。
/ter\b/	可匹配 chapter，而不能匹配 terminal。
/\Bapt/	可匹配 chapter，而不能匹配 aptitude。
/Windows(?=95 |98 |NT)/	可匹配 Windows95 或 Windows98 或 WindowsNT，当找到一个匹配后，从 Windows 后面开始进行下一次的检索匹配。
/^\s*$/	匹配空行。
/\d{2}-\d{5}/	验证由两位数字、一个连字符再加 5 位数字组成的 ID 号。
/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/	匹配 HTML 标记。

9.4. 常用正则表达式

9.4.1. 校验数字的表达式

数字：**^[0-9]*$**
n 位的数字：**^\d{n}$**
至少 n 位的数字：**^\d{n,}$**

m-n 位的数字：**^\d{m,n}$**
零和非零开头的数字：**^(0|[1-9][0-9]*)$**
非零开头的最多带两位小数的数字：**^([1-9][0-9]*)+(\.[0-9]{1,2})?$**
带 1-2 位小数的正数或负数：**^(\-)?\d+(\.\d{1,2})$**
正数、负数、和小数：**^(\-|\+)?\d+(\.\d+)?$**
有两位小数的正实数：**^[0-9]+(\.[0-9]{2})?$**
有 1~3 位小数的正实数：**^[0-9]+(\.[0-9]{1,3})?$**
非零的正整数：**^[1-9]\d*$ 或 ^([1-9][0-9]*){1,3}$ 或 ^\+?[1-9][0-9]*$**
非零的负整数：**^\-[1-9][]0-9"*$ 或 ^-[1-9]\d*$**
非负整数：**^\d+$ 或 ^[1-9]\d*|0$**
非正整数：**^-[1-9]\d*|0$ 或 ^((-\d+)|(0+))$**
非负浮点数：**^\d+(\.\d+)?$ 或 ^[1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0$**
非正浮点数：**^((-\d+(\.\d+)?)|(0+(\.0+)?))$** 或 **^(-([1-9]\d*\.\d*|0\.\d*[1-9]\d*))|0?\.0+|0$**
正浮点数：**^[1-9]\d*\.\d*|0\.\d*[1-9]\d*$** 或
^(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*))$
负浮点数：**^-([1-9]\d*\.\d*|0\.\d*[1-9]\d*)$** 或
^(-(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*)))$
浮点数：**^(-?\d+)(\.\d+)?$ 或 ^-?([1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0)$**

9.4.2. 校验字符的表达式

- 汉字：**^[\u4e00-\u9fa5]{0,}$**
- 英文和数字：**^[A-Za-z0-9]+$ 或 ^[A-Za-z0-9]{4,40}$**
- 长度为 3-20 的所有字符：**^.{3,20}$**
- 由 26 个英文字母组成的字符串：**^[A-Za-z]+$**
- 由 26 个大写英文字母组成的字符串：**^[A-Z]+$**
- 由 26 个小写英文字母组成的字符串：**^[a-z]+$**
- 由数字和 26 个英文字母组成的字符串：**^[A-Za-z0-9]+$**
- 由数字、26 个英文字母或者下划线组成的字符串：**^\w+$ 或 ^\w{3,20}$**
- 中文、英文、数字包括下划线：**^[\u4E00-\u9FA5A-Za-z0-9_]+$**
- 中文、英文、数字但不包括下划线等符号：**^[\u4E00-\u9FA5A-Za-z0-9]+$ 或 ^[\u4E00-\u9FA5A-Za-z0-9]{2,20}$**
- 可以输入含有^%&',;=?$\"等字符：**[^%&',;=?$\x22]+**
- 禁止输入含有~的字符：**[^~\x22]+**

9.4.3. 特殊需求表达式

- Email 地址：**^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$**
- 域名：**[a-zA-Z0-9][-a-zA-Z0-9]{0,62}(/.[a-zA-Z0-9][-a-zA-Z0-9]{0,62})+/.?**
- InternetURL：**[a-zA-z]+://[^\s]* 或 ^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]*)?$**
- 手机号码：**^(13[0-9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9])\d{8}$**
- 电话号码("XXX-XXXXXXX"、"XXXX-XXXXXXXX"、"XXX-XXXXXXX"、"XXX-XXXXXXXX"、"XXXXXXX"和"XXXXXXXX)：**^(\(\d{3,4}-)|\d{3.4}-)?\d{7,8}$**
- 国内电话号码(0511-4405222、021-87888822)：**\d{3}-\d{8}|\d{4}-\d{7}**
- 电话号码正则表达式（支持手机号码，3-4 位区号，7-8 位直播号码，1 - 4 位分机号）：**((\d{11})|^((\d{7,8})|(\d{4}|\d{3})-(\d{7,8})|(\d{4}|\d{3})-(\d{7,8})-(\d{4}|\d{3}|\d{2}|\d{1})|(\d{7,8})-(\d{4}|\d{3}|\d{2}|\d{1}))$)**
- 身份证号(15 位、18 位数字)，最后一位是校验位，可能为数字或字符 X：**(^\d{15}$)|(^\d{18}$)|(^\d{17}(\d|X|x)$)**
- 帐号是否合法(字母开头，允许 5-16 字节，允许字母数字下划线)：**^[a-zA-Z][a-zA-Z0-9_]{4,15}$**
- 密码(以字母开头，长度在 6~18 之间，只能包含字母、数字和下划线)：**^[a-zA-Z]\w{5,17}$**
- 强密码(必须包含大小写字母和数字的组合，不能使用特殊字符，长度在 8-10 之间)：**^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])[a-zA-Z0-9]{8,10}$**
- 强密码(必须包含大小写字母和数字的组合，可以使用特殊字符，长度在 8-10 之间)：**^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$**
- 日期格式：**^\d{4}-\d{1,2}-\d{1,2}**
- 一年的 12 个月(01～09 和 1～12)：**^(0?[1-9]|1[0-2])$**
- 一个月的 31 天(01～09 和 1～31)：**^((0?[1-9])|((1|2)[0-9])|30|31)$**

- xml 文件：**^([a-zA-Z]+-?)+[a-zA-Z0-9]+\\.[x|X][m|M][l|L]$**
- 中文字符的正则表达式：**[\u4e00-\u9fa5]**

- 双字节字符：**[^\x00-\xff] (包括汉字在内，可以用来计算字符串的长度(一个双字节字符长度计 2，ASCII 字符计 1))**
- 空白行的正则表达式：**\n\s*\r (可以用来删除空白行)**
- HTML 标记的正则表达式：**<(\S*?)[^>]*>.*?|<.*? /> (首尾空白字符的正则表达式：^\s*|\s*$或(^\s*)|(\s*$) (可以用来删除行首行尾的空白字符(包括空格、制表符、换页符等等)，非常有用的表达式)**
- 腾讯 QQ 号：**[1-9][0-9]{4,} (腾讯 QQ 号从 10000 开始)**
- 中国邮政编码：**[1-9]\d{5}(?!\d) (中国邮政编码为 6 位数字)**
- IP 地址：**((?:(?:25[0-5]|2[0-4]\\d|[01]?\\d?\\d)\\.){3}(?:25[0-5]|2[0-4]\\d|[01]?\\d?\\d))**

9.4.4. 钱的输入格式

有四种钱的表示形式我们可以接受:"10000.00" 和 "10,000.00", 和没有 "分" 的 "10000" 和 "10,000"：

^[1-9][0-9]*$

这表示任意一个不以 0 开头的数字,但是,这也意味着一个字符"0"不通过,所以我们采用下面的形式：

^(0|[1-9][0-9]*)$

一个 0 或者一个不以 0 开头的数字.我们还可以允许开头有一个负号：**^(0|-?[1-9][0-9]*)$**

这表示一个0或者一个可能为负的开头不为0的数字.让用户以0开头好了.把负号的也去掉,因为钱总不能是负的吧。

下面我们要加的是说明可能的小数部分：**^[0-9]+(.[0-9]+)?$**

必须说明的是,小数点后面至少应该有 1 位数,所以"10."是不通过的,但是 "10" 和 "10.2" 是通过的：

^[0-9]+(.[0-9]{2})?$

这样我们规定小数点后面必须有两位,如果你认为太苛刻了,可以这样：**^[0-9]+(.[0-9]{1,2})?$**

这样就允许用户只写一位小数.下面我们该考虑数字中的逗号了,我们可以这样：

^[0-9]{1,3}(,[0-9]{3})*(.[0-9]{1,2})?$

1 到 3 个数字,后面跟着任意个 逗号+3 个数字,逗号成为可选,而不是必须：

^([0-9]+|[0-9]{1,3}(,[0-9]{3})*)(.[0-9]{1,2})?$

备注：这就是最终结果了,别忘了"+"可以用"*"替代如果你觉得空字符串也可以接受的话(奇怪,为什么?)最后,别忘了

在用函数时去掉去掉那个反斜杠,一般的错误都在这里

10. Python 练习实例

10.1. 数组倒序

将一个列表顺序颠倒
实现代码：

```
arr = [9, 6, 5, 4, 1]
reverseList = list(reversed(arr))
print("反转前：", arr)
print("反转后：", reverseList)
```

结果：

```
反转前：  [9, 6, 5, 4, 1]
反转后：  [1, 4, 5, 6, 9]
```

10.2. 有限排列问题

题目：有四个数字：1、2、3、4，能组成多少个互不相同且无重复数字的三位数？各是多少？
实现：

```
#coding=utf-8
#题目：有四个数字：1、2、3、4，能组成多少个互不相同且无重复数字的三位数？各是多少？

z_ = ["%d%d%d" % (x, y, z) for x in range(1, 5) for y in range(1, 5) for z in range(1, 5) if
      (x != y) and (x != z) and (y != z)]
print(z_)
```

10.3. 使用正则表达式提取字符串中的 URL

```
import re

def find_url(string):
    # findall() 查找匹配正则表达式的字符串
    url = re.findall('https?://(?:[-\w.]|(?:%[\da-fA-F]{2}))+', string)
    return url

string = 'baidu 的网页地址为：http://www.baidu.com，Google 的网页地址为：
https://www.google.com,http://xiaoxiaoming.xyz/'
```

```
print("Urls: ", find_url(string))
```

10.4. 数组顺序移动

题目：有 n 个整数，将最后 m 个数移动到最前面。
实现代码：

```
# coding=utf-8

a = [1, 2, 3, 4, 5, 6, 7]
def move(array, m):
    return array[-m:] + array[:len(a) - m]
print(move(a, 3))
```

将数组 arr 的前面 d 个元素翻转到数组尾部:

```
arr = [1, 2, 3, 4, 5, 6, 7]

# 将长度为 n 的 数组 arr 的前面 d 个元素翻转到数组尾部。
def leftRotate(arr, d):
    return arr[d - len(arr):] + arr[:d]

print(leftRotate(arr, 2))
```

10.5. 斐波那契数列

斐波那契数列（Fibonacci sequence），又称黄金分割数列，指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……。
在数学上，费波那契数列是以递归的方法来定义：
F(0)=0(n=0)
F(1)=1(n=1)
F(n)=F(n-1)+F(n-2)(n>=2，n∈N*)

递归实现：

```
# coding=utf-8

def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

# 输出了第 10 个斐波那契数列
print("斐波那契数列：")
for i in range(10):
    print(fib(i), end=",")
```

循环实现：

```
# coding=utf-8

# 第一和第二项
n1, n2 = 0, 1
n = 10

print("斐波那契数列：")
print(n1, ",", n2, end=" , ")
for i in range(n - 2):
    # 更新值
    n1, n2 = n2, n1 + n2
    print(n2, end=" , ")
```

10.6. 获取指定期间内的阿姆斯特朗数

如果一个 n 位正整数等于其各位数字的 n 次方之和,则称该数为阿姆斯特朗数。 例如 1^3 + 5^3 + 3^3 = 153。
1000 以内的阿姆斯特朗数： 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407。

```
# coding=utf-8

# 1000 以内的阿姆斯特朗数： 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407
for num in range(1, 1000 + 1):
    # 初始化 sum
    sum = 0
    n = len(str(num))
    temp = num
    while temp > 0:
        digit = temp % 10
        sum += digit ** n
        temp //= 10
    if num == sum:
        print(num, end=" ")
```

10.7. 二分搜索

用 2 分法快速在有序列表中找到指定元素在列表中的角标，若没有找到则返回-插入点-1(<0)的值
例如：
在有序列表 list1 中[1, 3, 8, 12, 23, 31, 37, 42, 48, 58]中查找值为 8 的记录的。
在有序列表 list1 中[1, 3, 8, 12, 23, 31, 37, 42, 48, 58]中查找值为 9(不存在，以负数形式返回插入点的位置)的记录。

实现代码：

```
# coding=utf-8
```

```
# 自定义函数，实现二分查找，并返回查找结果
def binarySearch(arr, destValue):
    low = 0
    high = len(arr) - 1
    mid = (low + high) // 2
    while low <= high:
        if arr[mid] == destValue:
            return mid
        elif arr[mid] > destValue:
            high = mid - 1
        else:
            low = mid + 1
        mid = (low + high) // 2
    # 未找到返回-插入点-1
    return -mid - 1


list1 = [1, 3, 8, 12, 23, 31, 37, 42, 48, 58]
print(list(enumerate(list1)))
print(binarySearch(list1, 8), 8)
print(binarySearch(list1, 9), 9)
```

结果：

```
[(0, 1), (1, 3), (2, 8), (3, 12), (4, 23), (5, 31), (6, 37), (7, 42), (8, 48), (9, 58)]
2 8
-3 9
```

10.8. 计算指定范围的素数

题目：判断 101-200 之间有多少个素数，并输出所有素数。

```
# coding=utf-8

import math

result = []
for num in range(101, 200 + 1):
    if num > 1:
        for i in range(2, int(math.sqrt(num))+1):
            if (num % i) == 0:
                break
        else:
            result.append(num)
print("共有%d 个素数，素数列表为：" % len(result), result)
```

结果：

```
共有 21 个素数，素数列表为： [101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199]
```

10.9. 约瑟夫环

题目：有 n 个人围成一圈，顺序排号。从第一个人开始报数（从 1 到 m 报数），凡报到 m 的人退出圈子，直到船上仅剩 r 人为止,问都有哪些编号的人下船了呢？。
假设 n=30，m=9，r=15
实现代码：

```
# coding=utf-8

n, m, r = 30, 9, 15
circle = list(range(1, n + 1))
# index+1 代表当前报数的人在剩余报数人群中的编号
index = 0
result = []
while len(circle) > r:
    index += m-1
    if (index >= len(circle)):
        index -= len(circle)
    result.append(circle.pop(index))
print("报到%d 的人的出列顺序：" % m, result)
```

结果：

```
报到 9 的人的出列顺序： [9, 18, 27, 6, 16, 26, 7, 19, 30, 12, 24, 8, 22, 5, 23]
```

10.10. 单词计数

在 D:\hdfs\wordcount\in 有一些文本文件，要求统计出这些文本文件所有出现的单词的出现次数
代码实现：

```
# coding=utf-8

import os

dirPath = "/hdfs/wordcount/in"

map = {}
for file in os.listdir(dirPath):
    filename = os.path.join(dirPath, file)
    with open(filename) as fo:
        for line in fo:
            for word in line.strip().split(" "):
                map[word] = map.get(word, 0) + 1
print(map)
```

运行结果示例：

```
{'apple': 3, 'cdh': 2, 'dest': 4, 'firend': 2, 'english': 1, 'girl': 2, 'gift': 1, 'hit': 1, 'hello': 8, 'world': 3, 'you': 6, 'giles': 1, 'me': 9,
```

'to': 3, 'ahe': 1, 'jjio': 1, 'hjk': 1, 'fhg': 1}

有一篇英文文章要求统计每个单词出现的次数，将出现次数最多前十个个单词写入到 mysql 数据库中:

```
# coding=utf-8

import re, sys

dict1 = {}
with open(r"D:\hdfs\wordcount\article.txt", encoding="utf-8") as fo:
    for line in fo:
        words = re.compile("\W+").split(line)
        for word in words:
            if word and len(word) > 1:
                dict1[word] = dict1.get(word, 0) + 1

items = sorted(dict1.items(), key=lambda x: x[1], reverse=True)
items = items[:9]

import pymysql

db = pymysql.connect("localhost", "root", "123456", "bigdata")
cursor = db.cursor()
try:
    sql = r"insert into wordcount Values('%s',%d)"
    for k, v in items:
        sql_curr = sql % (k, v)
        print(sql_curr)
        cursor.execute(sql_curr)
    db.commit()
except BaseException as v:
    sys.stderr.write(str(v))
    db.rollback()
db.close()
```

10.11. python 实现常见的排序算法

10.11.1. 选择排序

```
'''
选择排序（Selection sort）工作原理如下:
首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，
然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
以此类推，直到所有元素均排序完毕。
'''
```

```
def selectionSort(arr):
    for i in range(len(arr) - 1):
        min = i
        for j in range(i + 1, len(arr)):
            if (arr[min] > arr[j]): min = j
        arr[i], arr[min] = arr[min], arr[i]
    return arr
```

10.11.2. 冒泡排序

```
'''
冒泡排序（Bubble Sort）：
对于未排序序列，循环比较相邻两个元素，顺序错误就交换位置
一趟排序下来会将最大（最小）元素浮动到最末端，
将除了最末端元素以外的序列作为未排序序列重复上面步骤
'''

def bubbleSort(arr):
    for i in range(len(arr) - 1):
        for j in range(len(arr) - i - 1):
            if (arr[j] > arr[j + 1]):
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

10.11.3. 快速排序

```
'''
快速排序(Quick Sort)
使用分治法（Divide and conquer）策略来把一个序列（list）分为较小和较大的 2 个子序列，然后递归地排序两个子序列。
步骤:
1.挑选基准值：从数列中挑出一个元素，称为"基准"（pivot）；
2.分割：所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（与基准值相等的数可以到任何一边）
在这个分割结束之后，对基准值的排序就已经完成;

然后对分割好的大小两个子序列再递归进行上述过程，直到子序列长度小于等于 1.

基准值选取方法有很多，这里直接选择待排序序列的起始元素作为基准值。
'''

def partition(arr, low, high):
    pivot = arr[low]
    while low < high:

        # 从右向左查找比基准值小的位置
```

```
        while low < high and arr[high] >= pivot: high -= 1
        arr[low] = arr[high]

        # 从左向右查找比基准值大的位置

        while low < high and arr[low] <= pivot: low += 1
        arr[high] = arr[low]
    arr[low] = pivot
    return low


# 快速排序函数

def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

10.11.4. 插入排序

```
'''
插入排序（Insertion Sort）
工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。
'''


def insertionSort(arr):
    for i in range(1, len(arr)):

        key = arr[i]  # 待插入元素，arr[0:i-1]作为有序序列


        j = i - 1  # j 指向有序序列的末端

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

10.11.5. 希尔排序

```
'''
希尔排序（Shell Sort），也称递减增量排序算法，是基于插入排序的改进版本。
基本思想是：
先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序
待整个序列中的记录"基本有序"时,再对全体记录进行依次直接插入排序。
'''
```

```
def shellSort(arr):
    incr = len(arr) // 2
    while incr > 0:

        # i 表示待插入元素角标，arr[0:i:incr]将作为有序序列

        #例如有序序列角标为{0，incr,2*incr,...,i-incr},待插入元素角标为 i,{n*incr}

        for i in range(incr, len(arr), incr):

            key = arr[i] # 待插入元素，arr[0:i:incr]作为有序序列


            j = i - incr # j 指向有序序列的末端

            while j >= 0 and arr[j] > key:
                arr[j + incr] = arr[j]
                j -= incr
            arr[j + incr] = key
        incr //= 2
```

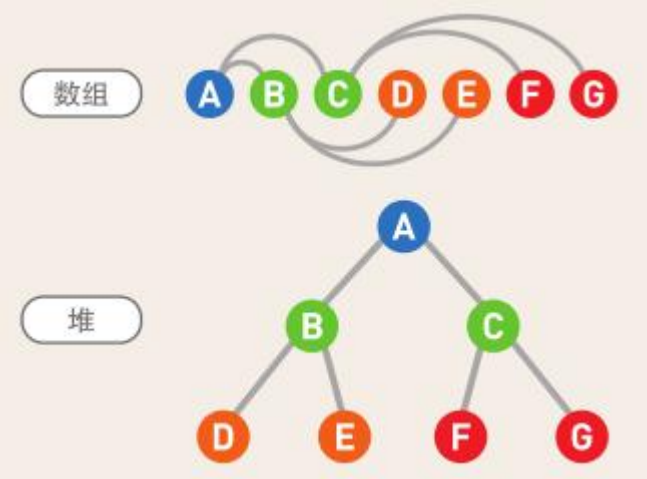
10.11.6. 归并排序

```
'''

归并排序（Merge sort），是创建在归并操作上的一种有效的排序算法。

该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。



分治法:

分割：递归地把当前序列平均分割成两半。

集成：在保持元素顺序的同时将上一步得到的子序列集成到一起（归并）。

'''


def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # 创建临时数组
    L = [0] * (n1)
    R = [0] * (n2)

    # 拷贝数据到临时数组 arrays L[] 和 R[]
    for i in range(0, n1):
        L[i] = arr[l + i]
```

```
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # 归并临时数组到 arr[l..r]
    i = 0  # 初始化第一个子数组的索引
    j = 0  # 初始化第二个子数组的索引
    k = l  # 初始归并子数组的索引

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # 拷贝 L[] 的保留元素
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # 拷贝 R[] 的保留元素
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1


def mergeSort(arr, l, r):
    if l < r:
        m = (l + r) // 2

        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)
```

10.11.7. 堆排序

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。堆排序可以说是一种利用堆的概念来排序的选择排序。




```
'''
堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。
堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：
即子结点的键值或索引总是小于（或者大于）它的父节点。
堆排序可以说是一种利用堆的概念来排序的选择排序。
'''

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1  # left = 2*i + 1
    r = 2 * i + 2  # right = 2*i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]  # 交换

        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)
    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # 一个个交换元素

    for i in range(n - 1, 0, -1):

        arr[i], arr[0] = arr[0], arr[i]  #取出最大堆堆顶元素 arr[0]


        heapify(arr, i, 0) #重新生成最大堆
```

10.12. 正整数分解质因数

题目：将一个正整数分解质因数。例如：输入 90,打印出 90=2*3*3*5。

程序分析：对 n 进行分解质因数，应先找到一个最小的质数 k，然后按下述步骤完成：

- (1)如果这个质数恰等于 n，则说明分解质因数的过程已经结束，打印出即可。
- (2)如果 n<>k，但 n 能被 k 整除，则应打印出 k 的值，并用 n 除以 k 的商,作为新的正整数你 n,重复执行第一步。
- (3)如果 n 不能被 k 整除，则用 k+1 作为 k 的值,重复执行第一步。

代码实现：

```
# coding=utf-8
import math

def getMaxRangePrimeNumList(maxNum):
    result = []
    for num in range(2, maxNum + 1):
        if num > 1:
            for i in range(2, int(math.sqrt(num)) + 1):
                if (num % i) == 0:
                    break
            else:
                result.append(num)
    return result

def getPrimeFactorNumList(n):
    prime_num_list = getMaxRangePrimeNumList(n)
    result = []
    while n != 1:  # 循环保证递归
        for prime_num in prime_num_list:
            if n % prime_num == 0:
                n /= prime_num  # n 等于 n/index
                result.append(prime_num)
                break
    return result

def primeFactorization(n):
    print(n, '=', end=" ")
    if n == 1:
        print(1)
    else:
        primeFactorNumList = getPrimeFactorNumList(n)
        print("*".join(map(lambda x: str(x), primeFactorNumList)))

primeFactorization(90)
primeFactorization(1008700)
primeFactorization(54)
```

结果：

```
90 = 2*3*3*5
1008700 = 2*2*5*5*7*11*131
```

54 = 2*3*3*3

10.13. 最大公共字符串

写一个方法，要求能获取 2 个字符串的所有的最大相同子串，返回一个集合
例如：getMaxSubString("daccedbccfuhcfedjjfjba","dabccfedfjba")
运行结果为：[bccf, cfed, fjba]

```
# coding=utf-8

'''
写一个方法，要求能获取 2 个字符串的所有的最大相同子串，返回一个集合
例如：getMaxPublicSubString("daccedbccfuhcfedjjfjba","dabccfedfjba")
运行结果为：[bccf, cfed, fjba]
'''
'''
思路：选出长度最大和最小的字符串
最小的字符串，从完整字符串开始，循环长度-1 截取指定长度的字符串
遍历这些截取出来的字符串，判断最大字符串中是否包含这个字符串，
若包含，则这个字符串是最大公共字符串
'''

def getMaxPublicSubString(str1, str2):
    # 三元表达式语法：为真时的结果 if 判定条件 else 为假时的结果
    maxstr = str1 if len(str1) > len(str2) else str2
    minstr = str2 if maxstr == str1 else str1
    result = []
    flag = False
    # i 表示要截断的长度,范围从 0 到 len(minstr)-1
    for i in range(len(minstr)):
        # 例如：字符串"abcd"被截断 2 个长度后，变成"ab","bc","cd",共需循环 3 次
        # 实际起始截断范围是[0:len(minstr)-i]，结束截断范围[i:len(minstr)],则定义一个范围在 0~i 范围的变量j
即可
        for j in range(i + 1):
            substr = minstr[j:len(minstr) - i + j]
            # 不等于-1 表示该子字符串存在于 maxstr 中
            if (maxstr.find(substr) != -1):
                flag = True
                result.append(substr)
        if flag:
            return result
    return result


print(getMaxPublicSubString("dabccfedfjba", "daccedbccfuhcfedjjfjba"))
```

10.14. 杨辉三角

杨辉三角有多种重要的性质：

每行端点与结尾的数为 1.

每个数等于它上方两数之和。

每行数字左右对称，由 1 开始逐渐变大。

第 n 行的数字有 n 项

要求输入指定高度 n，例如 n=10 打印如下形式的杨辉三角形状：

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

1 7 21 35 35 21 7 1

1 8 28 56 70 56 28 8 1

1 9 36 84 126 126 84 36 9 1

实现代码：

```
# coding=utf-8

def printYanghuiTriangle(n):
    # an 是一个长度为 n，值全部为 1 的列表
    an = [1] * n
    for i in range(n):
        for j in range(i - 1, 0, -1):
            an[j] += an[j - 1]
        print("\t".join(map(lambda x: str(x), an[0:i + 1])))


printYanghuiTriangle(10)
```

10.15. 字符串排序组合问题

编程列出一个字符串的全字符组合情况，原始字符串中没有重复字符

例如：

打印排列 permutation 情况：

"abc" "acb" "bac" "bca" "cab" "cba"

打印组合 combination 情况：

"a" "b" "c" "ab" "ac" "bc" "abc"

原始字符串是"abc"，打印得到下列所有排列组合情况

"a" "b" "c"

"ab" "bc" "ca" "ba" "cb" "ac"

"abc" "acb" "bac" "bca" "cab" "cba"

10.15.1. 字符串全排列问题

编程列出一个任意长度字符串的全字符排序情况，原始字符串中没有重复字符
例如：
原始字符串是"abc"：
　　打印所有的排列 permutation 情况：
　　"abc" "acb" "bac" "bca" "cab" "cba"
实现代码：

```
# coding=utf-8

def permutationStr(str):
    result = []

    def permutation(chars, begin):
        if begin == len(chars): result.append("".join(chars))
        for j in range(begin, len(chars)):
            chars[begin], chars[j] = chars[j], chars[begin]
            permutation(chars, begin + 1)
            chars[begin], chars[j] = chars[j], chars[begin]

    permutation(list(str), 0)
    return result


print(permutationStr("abc"))
```

结果：
['abc', 'acb', 'bac', 'bca', 'cba', 'cab']

10.15.2. 字符串全组合问题

编程列出一个任意长度字符串的全字符组合情况，原始字符串中没有重复字符
例如：
原始字符串是"abc"：
　　打印组合 combination 情况：
　　"a" "b" "c" "ab" "ac" "bc" "abc"
实现代码：

```
# coding=utf-8

def combinationStr(string):
    temp = []
    result = []

    def combination(chars, pos, num):
```

```
        if num == 0:
            result.append("".join(temp))
            return
        if pos == len(chars):
            return
        temp.append(chars[pos])
        combination(chars, pos + 1, num - 1)
        temp.pop(len(temp) - 1)
        combination(chars, pos + 1, num)

    chars = list(string)
    for i in range(1, len(chars) + 1):
        combination(chars, 0, i)
    return result

print(combinationStr("abc"))
```

10.15.3. 字符串全排列组合问题

编程列出一个任意长度字符串的全字符排列组合情况，原始字符串中没有重复字符

例如：

原始字符串是"abc"，打印得到下列所有排列组合情况

"a" "b" "c"

"ab" "bc" "ca" "ba" "cb" "ac"

"abc" "acb" "bac" "bca" "cab" "cba"

```
# coding=utf-8

'''
思路分析：
每行字符个数都是递增 ，下一行的字符是建立在上一行的基础上得到的
即在将第一行字符串长度限定为 1，第二行为 2....，
在第一行数据基础上{a,b,c}，创建第二行数据，
遍历字符串中所字符，并与第一行数据组合。注意每行字符串长度限制
1、先将原始字符串转换成字符数组 ch
2、将原始字符串每个字符添加到 list 集合中
3、遍历 list 集合用每个元素 str 去查找是否存在数组 ch 中的元素，如果 ch 中的字符 c 没有被 str 找到则用 str+c 作为新集合的值返回;
4、遍历新集合重复 3 步骤
'''


def permComStr(chars):
    # 将一个 list 用 chars 衍生，例如列表['a', 'b', 'c']会衍生出['ab', 'ac', 'ba', 'bc', 'ca', 'cb']
    def getDeriveList(srcList):
        if len(srcList) == 0: return list(chars)
        newList = []
        for str in srcList:
```

```
            for c in chars:
                if str.find(c) == -1:
                    newList.append(str + c)
        return newList

    # 初始化用于衍生的 list
    deriveList = []
    # 共需衍生 len(chars)次
    for i in range(len(chars)):
        # 衍生 list
        deriveList = getDeriveList(deriveList)
        print(" ".join(deriveList))

permComStr("abcd")
```

运行结果：

```
a b c d
ab ac ad ba bc bd ca cb cd da db dc
abc abd acb acd adb adc bac bad bca bcd bda bdc cab cad cba cbd cda cdb dab dac dba dbc dca dcb
abcd abdc acbd acdb adbc adcb bacd badc bcad bcda bdac bdca cabd cadb cbad cbda cdab cdba dabc dacb dbac dbca dcab
dcba
```

10.16. 螺旋矩阵

10.16.1. 外螺旋矩阵

螺旋矩阵是指一个呈螺旋状的矩阵，它的数字由第一行开始到右边不断变大，向下变大，向左变大，向上变大，如此循环。

例如一个宽度为 10，高度为 6 的螺旋矩阵如下：

```
1   2   3   4   5   6   7   8   9   10
28  29  30  31  32  33  34  35  36  11
27  48  49  50  51  52  53  54  37  12
26  47  60  59  58  57  56  55  38  13
25  46  45  44  43  42  41  40  39  14
24  23  22  21  20  19  18  17  16  15
```

对 w*h 矩阵，最先访问最外层的 h * w 的矩形上的元素，接着再访问里面一层的 (h - 2) * (w - 2) 矩形上的元素……最后可能会剩下一些元素，组成一个点或一条线。

对第 i 个矩形（i=0, 1, 2 …），4 个顶点的坐标为：

```
(i, i) ------------------------------------------- (i, w - 1-i)
|                                                   |
|                                                   |
|                                                   |
(h-1-i, i) ---------------------------------- (h-1-i, w-1-i)
```

要访问该矩形上的所有元素，只须用 4 个 for 循环，每个循环访问一个点和一边条边上的元素即可。另外，要注意对最终可能剩下的 1 * k 或 k * 1 矩阵再做个特殊处理。

代码实现：

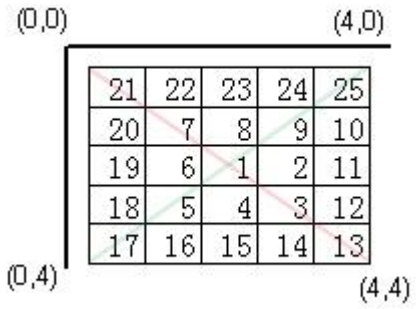
```
# coding=utf-8
import math


def outerSpiralMatrix(w, h):
    maxLen = w * h
    spiralMatrix = [([0] * w) for i in range(h)]
    floor = int(min(math.ceil(w / 2), math.ceil(h / 2)))
    num = 0
    for n in range(floor):
        for i in range(n, w - n):
            num += 1
            spiralMatrix[n][i] = num
        if num >= maxLen: break
        for i in range(n + 1, h - n):
            num += 1
            spiralMatrix[i][w - 1 - n] = num
        if num >= maxLen: break
        for i in range(w - 2 - n, n - 1, -1):
            num += 1
            spiralMatrix[h - 1 - n][i] = num
        if num >= maxLen: break
        for i in range(h - n - 2, n, -1):
            num += 1
            spiralMatrix[i][n] = num
        if num >= maxLen: break
    for matrix in spiralMatrix:
        print("\t".join(map(lambda x: str(x), matrix)))


outerSpiralMatrix(4, 5)
```

结果：

```
1	2	3	4
14	15	16	5
13	20	17	6
12	19	18	7
11	10	9	8
```

10.16.2. 内螺旋矩阵

打印一个如下图所示的矩阵



红线方程 y=x，绿线方程 y=-x+4，4 为矩形边长。
两条直线将区域分为四个部分，划分好每个区域的边界值，每个区域的坐标变化规律有四种，x++，y++，x--，y--
代码实现：

```
# coding=utf-8

def interSpiralMatrix(size):
    if (size % 2 != 1): size += 1
    spiralMatrix = [([0] * size) for i in range(size)]
    x, y, side = int(size / 2), int(size / 2), size - 1
    for i in range(1, size ** 2 + 1):
        spiralMatrix[y][x] = i
        if (y <= -x + side and y <= x):
            x += 1
        elif (-x + side < y and y < x):
            y += 1
        elif (x <= y and -x + side < y):
            x -= 1
        elif (x < y and y <= -x + side):
            y -= 1
    for matrix in spiralMatrix:
        print("\t".join(map(lambda x: str(x), matrix)))


interSpiralMatrix(5)
```

结果：

```
21	22	23	24	25
20	7	8	9	10
19	6	1	2	11
18	5	4	3	12
17	16	15	14	13
```

10.16.3. 双螺旋矩阵

双螺旋矩阵的定义如下，矩阵的最中心是 1，往上是 2，右拐 3，向下 4，然后依次 5、6，7...构成一条顺序增大的螺旋线，此外，如果从中心往下走的话，也是一条对称的螺旋线。

要求：给定一个矩阵维度 N，将其打印出来，示例如下。

```
25  14  15  16  17  18  19
24  13   6   7   8   9  20
23  12   5   2   3  10  21
22  11   4   1   4  11  22
21  10   3   2   5  12  23
20   9   8   7   6  13  24
19  18  17  16  15  14  25
```

代码实现：

```python
# coding=utf-8

def pairTnterSpiralMatrix(size):
    if (size % 2 != 1): size += 1
    arr = [([0] * size) for i in range(size)]
    max = (size * size + 1) // 2
    x, y = size // 2, size // 2
    arr[y][x] = 1
    y -= 1
    arr[y][x] = arr[size - 1 - y][size - 1 - x] = 2
    num, step = 2, 1
    while num < max:
        if step % 2 == 1:
            for i in range(step * 2 - 1):
                num += 1
                x += 1
                arr[y][x] = arr[size - 1 - y][size - 1 - x] = num
            for i in range(step * 2 + 1):
                num += 1
                y += 1
                arr[y][x] = arr[size - 1 - y][size - 1 - x] = num
                if (num == max): break
            step += 1
        else:
            for i in range(step * 2 - 1):
                num += 1
                x -= 1
                arr[y][x] = arr[size - 1 - y][size - 1 - x] = num
            for i in range(step * 2 + 1):
                num += 1
                y -= 1
                arr[y][x] = arr[size - 1 - y][size - 1 - x] = num
                if (num == max): break
            step += 1
    for matrix in arr:
        print("\t".join(map(lambda x: str(x), matrix)))


pairTnterSpiralMatrix(8)
```

结果：

```
41  26  27  28  29  30  31  32  33
```

```
40  25  14  15  16  17  18  19  34
39  24  13  6   7   8   9   20  35
38  23  12  5   2   3   10  21  36
37  22  11  4   1   4   11  22  37
36  21  10  3   2   5   12  23  38
35  20  9   8   7   6   13  24  39
34  19  18  17  16  15  14  25  40
33  32  31  30  29  28  27  26  41
```

10.17. 取各栏目粉丝量最多的用户

有一个微博网站有很多栏目，每一个栏目都有几万用户，每个用户会有很多的粉丝
要求取出各栏目粉丝量最多前 2 个的用户
样本数据

```
医疗 user1 user8397
艺术 user5 user6766
教育 user7 user9601
艺术 user5 user8036
法律 user7 user3953
艺术 user1 user1888
教育 user2 user5185
法律 user1 user5051
音乐 user6 user4751
法律 user6 user1204
法律 user7 user9517
教育 user3 user898
医疗 user6 user9067
音乐 user3 user2010
法律 user1 user6622
```

实现代码：

```python
# coding=utf-8

dict1 = {}
with open(r"D:\hdfs\logs\weibo.log", encoding="utf-8") as fo:
    for line in fo:
        if not line: break
        fields = line.strip().split(" ")
        column, user, fensi = fields
        usercount = dict1.setdefault(column, {})
        usercount[user] = usercount.get(user, 0) + 1

for k, v in dict1.items():
    usercount = sorted(v.items(), key=lambda x: x[1], reverse=True)
    usercount = usercount[0:2]
    print(k, usercount)
```

10.18. 共同好友

以下是 qq 的好友列表数据，冒号前是一个用，冒号后是该用户的所有好友（数据中的好友关系是单向的）

样本数据

```
A:B,C,D,F,E,O
B:A,C,E,K
C:F,A,D,I
D:A,E,F,L
E:B,C,D,M,L
F:A,B,C,D,E,O,M
G:A,C,D,E,F
H:A,C,D,E,O
I:A,O
J:B,O
K:A,C,D
L:D,E,F
M:E,F,G
O:A,H,I,J
```

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

实现代码：

```
# coding=utf-8
str = \
"""A:B,C,D,F,E,O
B:A,C,E,K
C:F,A,D,I
D:A,E,F,L
E:B,C,D,M,L
F:A,B,C,D,E,O,M
G:A,C,D,E,F
H:A,C,D,E,O
I:A,O
J:B,O
K:A,C,D
L:D,E,F
M:E,F,G
O:A,H,I,J"""

firend2user_dict = {}
for line in str.splitlines():
    user, firend_str = line.split(":", 2)
    for firend in firend_str.split(","):
        user_list = firend2user_dict.setdefault(firend, [])
```

```
        user_list.append(user)

result_dict = {}
for firend, user_list in firend2user_dict.items():
    for i in range(0, len(user_list) - 1):
        for j in range(i + 1, len(user_list)):
            user2user = "%s-%s" % (user_list[i], user_list[j])
            common_friend_list = result_dict.setdefault(user2user, [])
            common_friend_list.append(firend)

for user2user, common_friend_list in sorted(result_dict.items(), key=lambda x: x[0]):
    print(user2user, ",".join(common_friend_list))
```

```
A-B C,E
A-C D,F
A-D F,E
A-E B,C,D
A-F B,C,D,E,O
A-G C,D,F,E
A-H C,D,E,O
A-I O
A-J B,O
A-K C,D
A-L D,F,E
A-M F,E
B-C A
B-D E,A
B-E C
B-F C,E,A
B-G C,E,A
B-H C,E,A
B-I A
B-K C,A
B-L E
B-M E
B-O A
C-D F,A
C-E D
C-F D,A
C-G D,F,A
C-H D,A
C-I A
C-K D,A
C-L D,F
C-M F
C-O A,I
D-E L
D-F E,A
D-G F,E,A
```

D-H E,A
D-I A
D-K A
D-L F,E
D-M F,E
D-O A
E-F B,C,D,M
E-G C,D
E-H C,D
E-J B
E-K C,D
E-L D
F-G C,D,E,A
F-H C,D,E,O,A
F-I O,A
F-J B,O
F-K C,D,A
F-L D,E
F-M E
F-O A
G-H C,D,E,A
G-I A
G-K C,D,A
G-L D,F,E
G-M F,E
G-O A
H-I O,A
H-J O
H-K C,D,A
H-L D,E
H-M E
H-O A
I-J O
I-K A
I-O A
K-L D
K-O A
L-M F,E