Python 程序设计

序 言

教材说明

- 教材为 Python 编程语言及应用的基础教材;
- 教材分为 12 个章节，从零基础入门，全面介绍 Python 编程语言的基础知识，通过大量的实战案例和训练任务，让学习者快速掌握 Python 的编程知识;
- 教材通过对 Python 语言的对各类型数据文件操作、多线程以及数据采集领域的理论基础以及实际应用操作技巧的介绍，让读者快速掌握 Python 编程语言。

教材目标

- 掌握 Python 编程语言的运行及开发环境的搭建;
- 掌握 Python 编程语言的基础知识，如 数据类型、表达式、序列对象、控制流语句、函数编程以及 OOP;
- 全面认知 Python 编程思想;
- 掌握 Python 对常用文件及数据文件的读写操作;
- 掌握多线程编程的应用技术。

Python 官方文档

- Python 官方文档库 https://docs.python.org/3/
- Python 标准库中文文档.pdf

网络资料

- StackOverflow IT 领域内的“Facebook”https://stackoverflow.com
- Python GitHub 源码 https://github.com/python
- CSDN 国内知名的 IT 垂直领域的网站 https://www.csdn.net

目 录

第 01 章：Python 快速入门 13

1.1 Python 语言简介 13

1.1.1 语言发展历史 13

1.1.2 语言的定义 13

1.1.3 创始人小故事 14

1.1.4 Python 应用领域 14

1.1.5 Python 语言特点 15

1.1.6 Python 语言在行业的认知 16

1.1.7 实战任务：Python 在 Windows 运行环境的搭建 16

1.2 Python 工作原理及运行方式 19

1.2.1 Python 工作原理 19

1.2.2 运行方式一：脚本程序直接运行 20

1.2.3 运行方式二：解释器编译成中间文件 21

1.2.4 PVM 虚拟机原理 21

1.3 Python 在 Windows 系统中的开发环境搭建 23

1.3.1 Python 版本介绍 23

1.3.2 Pip 插件配置 24

1.3.3 Python 虚拟环境 27

1.3.4 pywin32 的安装配置 27

1.3.5 Python 开发 IDE 介绍 28

1.3.6 实战任务：VSCode 的安装配置 31

1.4 本章总结 34

第 02 章：Python 语法规范与数据类型 35

2.1 Python 基础语法 35

2.1.1 Python 编程模式 35

2.1.2 字符编码 36

2.1.3 标识符和保留字 39

2.1.4 语法格式规范 41

2.2 Python 注释及规范 44

2.2.1 注释及其他 44
2.2.2 引号使用说明 44
2.2.3 Python 注释 45
2.2.5 Python 空行 46
2.2.5 等待用户输入 47
2.2.6 print 屏幕输出 48
2.3 命令行参数 49
2.3.1 什么是命令行参数 49
2.3.2 实战任务：模拟用户个人信息注册 50
2.4 基本数据类型 51
2.4.1 变量基本概念 51
2.4.2 什么是变量？ 51
2.4.3 数据类型 51
2.4.4 标准数据类型 52
2.4.5 变量赋值 52
2.4.6 标准数据类型 54
2.5 本章总结 60

第 03 章：序列类型及运算符 62

3.1 序列对象（sequence） 62
3.1.1 序列与数组的区别 62
3.1.2 Python 中常用的序列对象 62
3.1.3 列表（List）类型 63
3.1.4 元组（Tuple）类型 67
3.1.5 字典（Dictionary）类型 68
3.1.6 集合（Set）类型 70
3.1.7 数据类型转换 71
3.2 运算符 71
3.2.1 Python 运算符 72
3.2.2 算数运算符 72
3.2.3 比较运算符 73
3.2.4 赋值运算符 74

3.2.5 位运算符 75
3.2.5 逻辑运算符 76
3.2.6 成员运算符 76
3.2.7 身份运算符 77
3.2.8 is 与 == 的区别 78
3.2.9 Python 运算符优先级 79
3.3 本章总结 80

第 04 章：控制流语句（I） 81

4.1 控制流语句 81
4.1.1 用途 81
4.1.2 条件控制语句介绍 81
4.1.3 循环控制语句介绍 82
4.2 条件控制语句 82
4.2.1 基本 if 语法介绍 82
4.2.2 if…else… 结构 82
4.2.3 多值判断 if…elif…else… 83
4.2.4 多值判断嵌套 if 结构 83
4.2.6 实战任务：模拟用户登录 84
4.3 循环控制语句 87
4.3.1 循环控制语句 87
4.3.2 循环控制语句的分类 87
4.3.3 循环控制语句中的关键字 88
4.3.4 While 循环控制语句 88
4.3.5 break 和 continue 90
4.3.6 无限循环 91
4.3.7 实战任务：菜单栏的生成 91
4.3.8 循环使用 while…else 语句 94
4.3.9 实战任务：用户信息录入 94
4.4 iterator 迭代器 96
4.4.1 迭代器的语法 96
4.4.2 实战任务：城市信息显示 97

4.5 断点调试 98
4.5.1 断点调试的意义 98
4.5.2 IDLE 进行断点调试 98
4.6 本章总结 99

第 05 章：控制流语句（II） 100
5.1 再看 range 数据类型 100
5.1.1 range 类型 100
5.1.2 range（）示例 100
5.2 For 循环语句 101
5.2.1 for 循环语句 101
5.2.2 for 循环使用说明 102
5.2.3 嵌套 for 循环 103
5.2.4 for…else…语句 103
5.2.5 实战任务：银行金额大写汉字转换 104
5.3 推导式 107
5.3.1 推导式介绍 107
5.3.2 列表推导式 108
5.3.3 字典推导式 109
5.3.4 集合推导式 109
5.4 异常处理 110
5.4.1 Python3 的错误和异常 110
5.4.2 异常 Except 111
5.4.3 异常的处理的关键字及语法结构 111
5.4.4 异常的处理 try…except 112
5.4.5 多 except 捕获 112
5.4.6 try…catch…else/finally 113
5.4.7 自定义条件手动抛出异常 114
5.5 本章总结 114

第 06 章：Python 函数编程 115
6.1 函数编程 115

6.1.1 函数基础知识 115
6.1.2 Python 函数 115
6.1.3 函数基本语法规则 115
6.1.4 自定义一个函数语法 116
6.1.5 函数的调用 116
6.1.6 return 关键字 117
6.1.7 参数的传递 117
6.2.1 可变（mutable）和不可变（immutable）对象 118
6.2 函数参数类型 120
6.2.1 关键字参数 121
6.2.2 必备参数 121
6.2.3.缺省参数 121
6.2.4 不定长参数 *args 和**kw 122
6.2 本章总结 124

第 07 章：Python 函数编程进阶 125

7.1 lambda 表达式 125
7.1.1 匿名函数概述 125
7.1.2 lambda 的表达式应用 126
7.2 变量作用域 127
7.2.1 全局变量和局部变量 127
7.2.2 递归 Recursion 128
7.2.3 示例 1：递归实现阶乘 129
7.2.4 示例 2：递归实现遍历指定文件或磁盘的检索 130
7.3 高阶函数 131
7.3.1 高阶函数定义 131
7.3.2 高阶函数的应用 131
7.4 装饰器 Decorator 132
7.4.1 装饰器的定义 132
7.4.2 装饰器的应用 132
7.5 闭包 Closure 134
7.5.1 闭包的定义 134

7.5.2 闭包的应用 134
7.7 生成器 Generator 135
7.7.1 生成器的定义 135
7.7.2 生成器的应用 135
7.7.3 生成器 yield 关键字 135
7.8 本章总结 136

第 08 章：面向对象编程基础 137

8.1 编程模式的变迁 137
8.1.1 概述 137
8.1.2 从面向过程开始 137
8.1.3 开启函数式编程模式 138
8.1.4 面向对象的编程模式 138
8.2 面向对象编程基础知识 138
8.2.1 面向对象的概念 138
8.2.2 对象的组成（状态+行为） 139
8.2.3 抽象过程 139
8.2.4 类 139
8.2.5 对象 140
8.2.6 面向对象的三大特征概述 140
8.3 Python 面向对象的快速实现 140
8.3.1 如何创建一个类 140
8.3.2 如何创建一个对象 141
8.4 类的方法 141
8.4.1 实例方法 instanceMethod 142
8.4.2 特殊的类实例方法 1-构造方法 143
8.4.3 特殊的类实例方法 2-实例对象输出 143
8.4.4 实例变量（成员属性） 144
8.4.5 实战任务：游戏人生 145
8.4.6 类方法 classmethod 148
8.4.7 类变量的定义及访问操作 149
8.5 静态方法 staticmethod 150

8.5.1 静态方法的定义 150
8.5.2 静态方法的定义及各种调用 150
8.5.3 静态方法可以访问实例变量 151
8.6 本章总结 152

第 09 章：Python 面向对象编程进阶 153

9.1 OOP 三大特征之一：封装 153
9.1.1 封装的概念 153
9.1.2 访问修饰符 153
9.1.3 访问修饰符的使用 154
9.1.4 访问修饰符在方法中的使用 155
9.1.5 @property 属性装饰器 156
9.1.6 对象属性的 setter 和 getter 156
9.2 OOP 三大特征之二：继承 159
9.2.1 继承 extends 159
9.2.2 继承的语法 160
9.2.3 多重继承 161
9.2.4 深度优先和广度优先 162
9.2.5 经典类 和 新式类 162
9.2.6 继承下的 super（）概述 162
9.2.7 当 super 遇到__init__ 163
9.3 OOP 三大特征之三：多态 163
9.3.1 多态的定义 163
9.3.2 “鸭子类型”的应用 164
9.4 本章总结 164

第 10 章：文件操作基础 165

10.1 文件操作基本原理 165
10.1.1 文件操作模块库 OS 165
10.1.2 文件的定义 165
10.1.3 文件路径 166
10.2 os 模块介绍 166

10.2.1 查看文件路径及文件名称 166
10.2.2 获取文件的创建、修改以及最后访问时间 168
10.2.3 判断文件夹或文件 169
10.2.4 获取文件夹的信息 170
10.2.5 获取文件夹的信息 os.walk()函数 170
10.2.6 实战任务：创建获取文件相关信息 171
10.2.7 文件夹的创建和删除 175
10.3 文件读写操作 175
10.3.1 读取文件 175
10.3.2 open 参数说明 176
10.3.3 使用 write( ) 函数实现文件写入 176
10.3.4 使用 read( ) 函数实现文件读取 178
10.3.5 二进制文件的读写操作 178
10.4 本章总结 179

第 11 章：常用数据文件操作 180

11.1 对象序列化 180
11.1.1 数据对象的序列化 180
11.1.2 序列化操作的意义 181
11.1.3 序列化应用的场景 181
11.2 Python 序列化操作 181
11.2.1 Pickle 模块介绍 181
11.2.2 Pickle 模块 API 函数应用 182
11.2.3 实战任务：OOP 对象序列化数据存储 183
11.3 Json 文件格式存储 186
11.3.1 JSON 数据 186
11.3.2 JSON 数据格式 187
11.3.3 json 模块 187
11.3.4 Python 中的 json 对象转换 188
11.3.4 JSON 模块 API 函数应用 188
11.4 CSV 文件格式存储 189
11.4.1 CSV 数据 190

11.4.2 csv 模块介绍 190
11.4.3 csv 文件读写操作 190
11.4.4 DictReader()/DictWriter() 191
11.5 Excel 文件读写操作 193
11.5.1 操作 excel 的模块介绍 193
11.5.2 Excel 写入操作模块 xlwt 模块 194
11.5.3 Excel 写入操作标准步骤 194
11.5.4 Excel 写入操作实现 194
11.5.4 Excel 读取操作 xlrd 模块 195
11.6 本章总结 195

第 12 章：多线程 196

12.1 多线程概述 196
12.1.1 什么是进程? 196
12.1.2 什么是线程? 197
12.1.3 什么是多线程? 198
12.1.4 多线程的优势 198
12.1.5 线程和进程之间的区别 199
12.2 Python 多线程实现 199
12.2.1 Python 实现多线程的方式 199
12.2.2 函数方式实现多线程 199
12.2.3 模块实现多线程_thread 200
12.2.4 模块实现多线程的语法 201
12.2.5 实战任务: 多线程实现“统筹方法” 202
12.3 线程同步 203
12.3.1 线程的工作状态 203
12.3.2 线程同步介绍 203
12.3.3 线程锁同步的使用方法 204
12.3.4 线程锁同步的流程分析 204
12.3.5 条件变量同步的介绍 206
12.4 实战任务: 生产者和消费者模式 206
12.4.1 设计模式说明 206

12.4.2 任务分解 207
12.4.3 任务实现 208
12.5 本章总结 210

第01章：Python快速入门

知识点

目标 1：了解 Python 语言的发展历史

目标 2：掌握 Python 语言在 Windows 操作系统中的运行环境搭建

目标 3：掌握 Python 语言开发工具的安装和配置

目标 4：编写第一个 Python 程序

技能点

实战任务 1：Python 语言在 Windows 操作系统下开发环境的搭建

实战任务 2：VSCode 开发工具搭建 Python 集成开发环境

实战任务 3：编写第一个 Python 程序

1.1 Python 语言简介

1.1.1 语言发展历史

自从 20 世纪 90 年代初 Python 语言诞生至今，它已被逐渐广泛应用于专业数据采集处理、数据科学计算分析以及自动化测试运维领域。

May 2017	May 2016	Change	Programming Language	Ratings	Change
1	1		Java	14.639%	-6.32%
2	2		C	7.002%	-6.22%
3	3		C++	4.751%	-1.95%
4	5	^	Python	3.548%	-0.24%
5	4	v	C#	3.457%	-1.02%
6	10	^^	Visual Basic .NET	3.391%	+1.07%
7	7		JavaScript	3.071%	+0.73%
8	12	^^	Assembly language	2.859%	+0.98%
9	6	v	PHP	2.693%	-0.30%
10	9	v	Perl	2.602%	+0.28%

图：2017 年 5 月 TIOBE 全球编程语言排行 Python 超越 C#语言升至第四名

1.1.2 语言的定义

Python 是一种 面向对象 的 解释型 计算机程序设计语言。是纯粹的自由软件， 源代码和解释器 CPython 遵循 GPL (GNU General Public License)协议。语法简洁清晰，特色之一是强制用空白符(white space)作为语句缩进 (标准四个空格) 。

1.1.3 创始人小故事

Python 由荷兰人 Guido van Rossum 于 1989 年发明，第一个公开发行版发行于 1991 年。1989 年圣诞节期间，在阿姆斯特丹，Guido 为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，做为 ABC 语言的一种继承。之所以选中 Python（大蟒蛇的意思）作为该编程语言的名字，是因为他是一个叫 Monty Python 的喜剧团体的爱好者。

1.1.4 Python应用领域

- 数据采集与处理 领域

使用 Requets/Urllib/Re 模块库实现典型的网络爬虫程序，采集各种结构化和非结构化数据，通过动态代理 Proxy 自动轮询，突破网站 403 反爬虫拦截机制。同时使用 Scrapy 或 BS4 企业级爬虫框架快速完成网络深维度自动探索采集。

- 数据计算与分析 领域

在数据处理方面使用强大的 NumPy / SciPy / Pandas 模块库实现数据规整化操作标准流程：采集->加载->清洗->转换->重塑；充分利用强大的 Pandas 模块库实现聚合与分组算法、时间序列算法等核心的数据分析计算；最终使用 Matpolitlib 模块库进行可视化数据呈现。

- 人工智能与机器学习 领域

使用 Scikit-Learn 模块库实现机器学习，使用 AIML 人工智能标记语言。掌握 Theano/Keras 模块库搭建各种深度学习模型，如自编码、循环神经网络、递归神经网络等。了解 Google 公司的 TensorFlow 人工智能系统。

- 自动化测试 领域

使用 Selenium2 模块库实现典型的网络模拟点击和虚拟操作，编写测试脚本完成对网站及应用的自动化测试，并进行测试日志存储记录和跟踪。结合相关平台和测试工具形成一整套的自动化测试标准流程和规范。

- 系统集成运维 领域

Fabric 模块库是基于 Python 实现的 SSH 命令行工具，简化了 SSH 的应用程序部署及系统管理任务，它提供了系统基础的操作组件，可以实现本地或远程 shell 命令，包括：命令执行、文件上传、下载及完整执行日志输出等功能。使用 Re 模块库对 Log 日志进行分析和处理。

- Web 互联网 领域

使用 Socket 模块库实现服务器及客户端编程，实现 TCP/UDP 的协议下的数据通信操作。利用 Django 框架快速实现网站开发，了解 web 服务器端框架 Flask/Tornado 在实际开发中的应用。

1.1.5 Python语言特点

- 简单易学

适合没有任何编程语言基础的人稍微看一下资料，就可以写出功能强大的程序。

- 开发效率高

很难像 Java 那样开发出完整的大型综合性网站或平台，但其起到画龙点睛的作用。同时也是一门典型的“胶水语言”，整合其他各种编程语言。

- 典型的工具语言

它是一门解释型编程语言，编译完毕后可直接运行，发现 Bug 后立即修改，剩下大量的编译时间。

■ 强大丰富的模块库

高度代码重用性，编写各种工具模块引用的系统工程中，丰富的模块强大到恐怖的地步，几乎无处不在适用于各种领域。

■ 优秀的跨平台

几乎所有的 Python 程序，都可以不加修改地运行在不同的操作系统平台。

1.1.6 Python 语言在行业的认知

“工具语言、胶水语言”很少像 Java 语言那样开发大型的企业级应用程序。Python 在设计上坚持了清晰划一的风格，易读、易维护，并且被大量用户所欢迎的、用途广泛的语言。丰富的模块库，使其八面玲珑。长期以来被作为工具语言 或 辅助编程语言，掌握 Java 编程语言的开发者中 70% 将 Python 作为第二辅助语言。

1.1.7 实战任务：Python在Windows运行环境的搭建

① Python3.6.5 64 位安装包的下载
② Windows 系统快速安装运行环境

任务目标

① Python3.6.5 64 位安装包的下载
② Windows 系统快速安装运行环境

■ Python3 的安装配置 综述

- 可以从官方网站 https://www.python.org/ 下载对应版本的 Python 安装包。
- Python 的安装和配置只需简单 3 步即可完成：

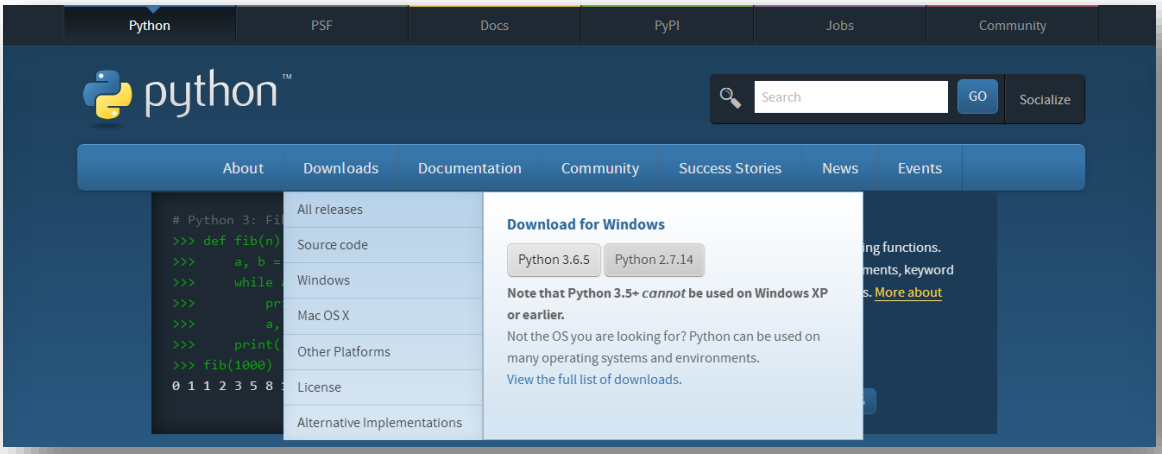
  Step1：双击安装 Python 运行开发环境
  Step2：在系统环境变量中验证 Python 指令集配置
  Step3：检查当前系统中默认的 Python 版本

■ Python 下载及安装

官网下载 Python 3.6.5

Step1：访问 Python 官方网站

网站地址：https://www.python.org/



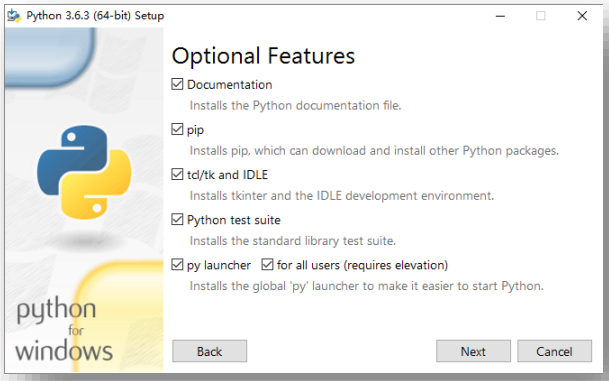
Step2：选择正确版本点击下载

■ 安装 Python 3.6.5

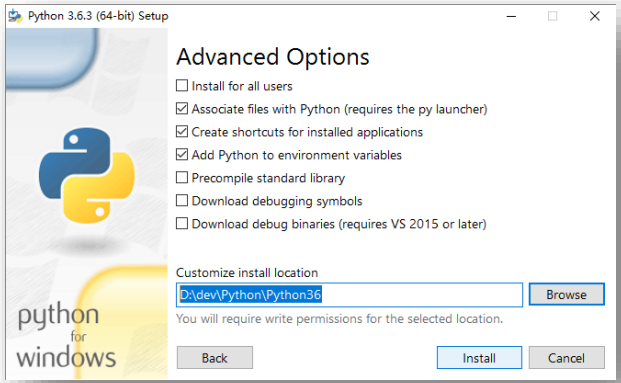
Step1:选择系统可用用户。选择 最先面的复选框 将 Python 环境配置到本地系统环境中；

Step2：确认默认安装的组件；

Step3：自定义安装路径。



■ 验证 Python 环境



Step1：打开 Windows 下的命令行工具。使用 win+R 快捷键启动 运行，输入 cmd 启动 命令行工具

Step2：使用 python 指令查看当前系统 Python 的环境版本

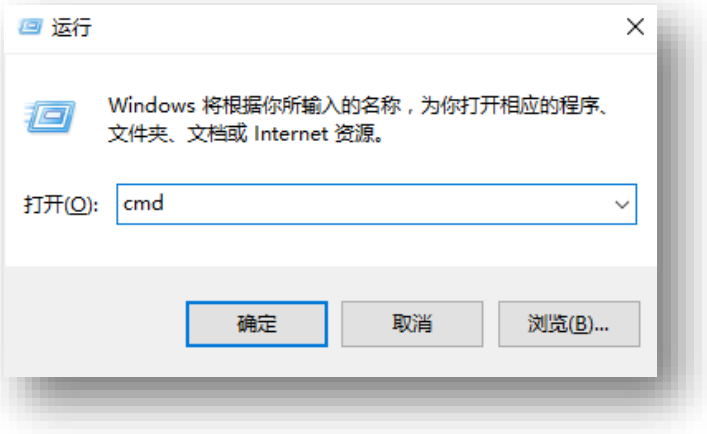
C:\ python -V

Python 3.6.5

若能正确显示 Python 版本号，代表安装正确。

否则报错“Python”不是内部或外部命令……

注意：命令参数 -V 为大写字母



■ 使用 命令行 工具测试

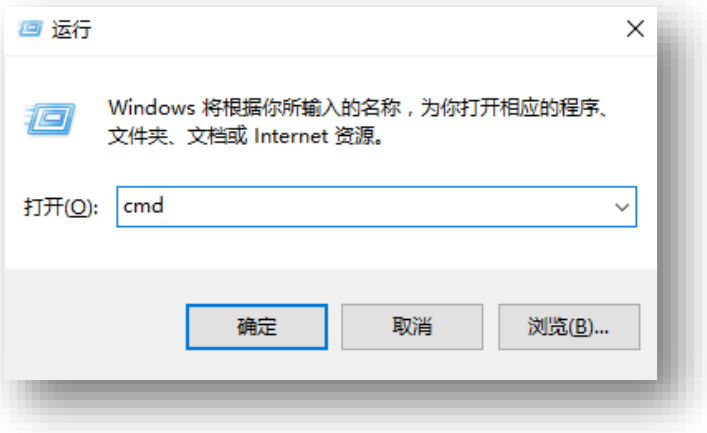
Step1：打开 Windows 下的命令行工具。使用 win+R 快捷键启动 运行，输入 cmd 启动 命令行工具。

Step2：使用 python 指令进入到 Python 编译环境。在 DOS 提示符下输入 python ，则进入到编译环境。

>>> 三箭头 代表 Python 提示符，可以输入指令或语句；

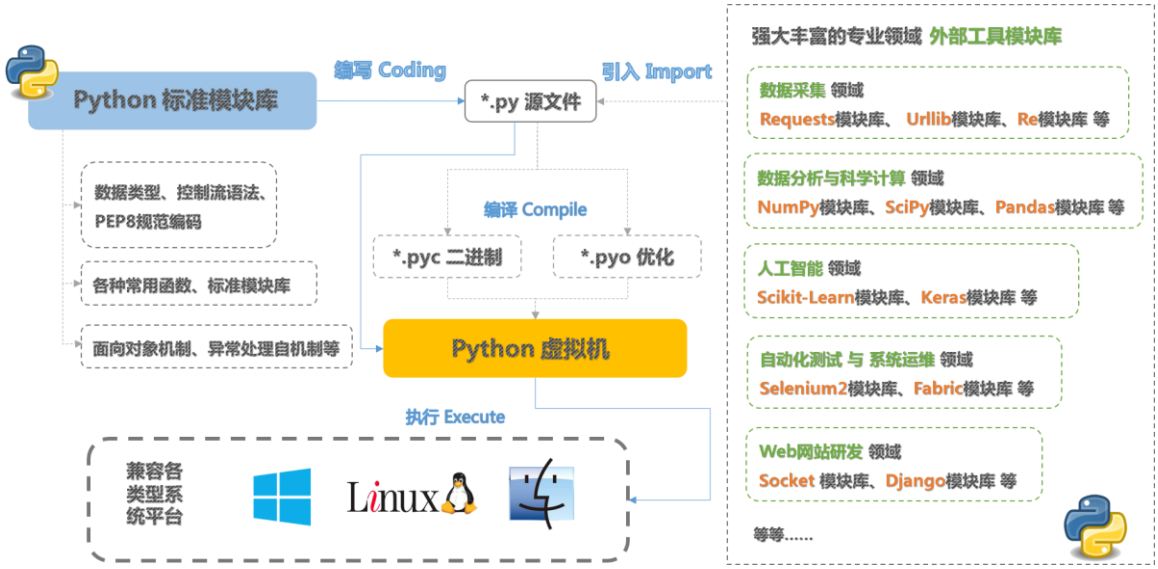
>>>print ‘……’代表 print 函数屏幕输出

可以使用 >>>exit( ) 函数退出 Python 编译环境



1.2 Python 工作原理及运行方式

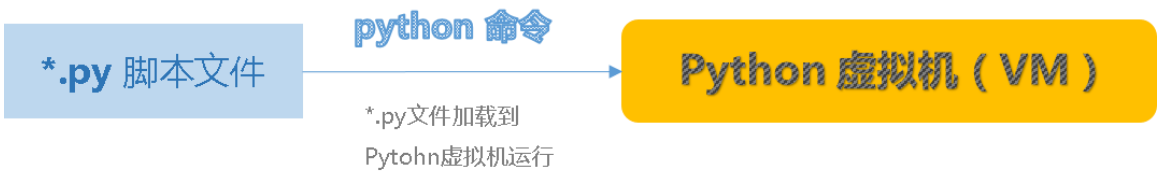
1.2.1 Python 工作原理



1.2.2 运行方式一：脚本程序直接运行

Python 语言是典型的脚本语言，通过解析器直接运行*.py 文件。所有 Python 脚本程序的后缀名都是以 *.py 结尾。

Python 脚本程序 *.py 编译过程



代码演示：创建编写 ch01-demo01.py 脚本源文件

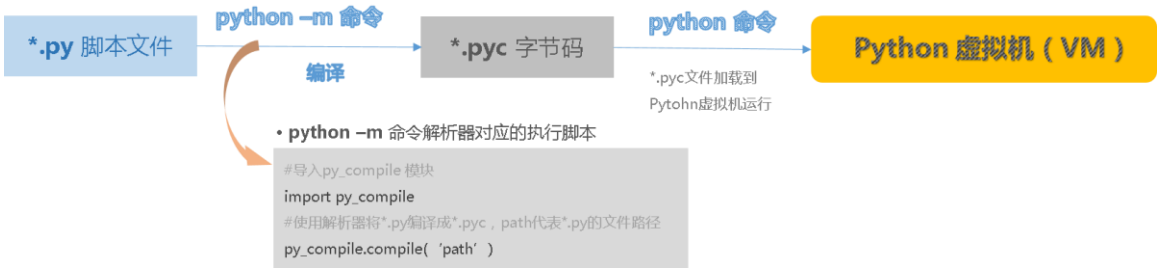
```
#-*- coding:utf-8 -*-
print(“你好，中软国际”)  # 终端输出
input(“<回车结束程序>”)  # 提示用户
```

运行输出： Dos 控制台使用 python 命令

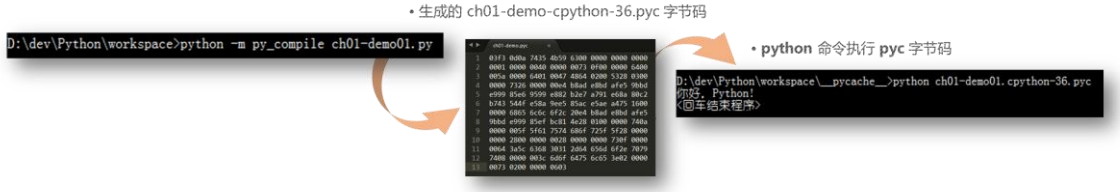
```
C:\>python d:\dev\Python\workspace\ch01-demo01.py
你好，Python!
<回车结束程序>
```

1.2.3 运行方式二：解释器编译成中间文件

pyc 脚本程序的编译执行过程



操作演示： Dos 控制台 *.py 编译成 *.pyc



■ 什么是 *.pyc 文件？

pyc 是一种二进制文件，是由 py 文件经过解释器编译后，在磁盘上生成的文件形式，是一种 byte code，py 文件变成 pyc 文件后，加载的速度有所提高，而且 pyc 是一种跨平台的字节码，是由 python 的解析器来执行的。pyc 的内容是跟 python 的版本相关的，不同版本编译后的 pyc 文件是不同的。

■ 什么时候需要 *.pyc 文件？

因为 py 文件是可以直接看到源码的，如果你是开发商业软件的话，不可能把源码也泄漏出去。所以就需要编译为 pyc 后，再发布出去。

1.2.4 PVM虚拟机原理

- PyCodeObject

pyc 字节码在 Python 虚拟机中对应的是 PyCodeObject 对象，虚拟机先把字节码封装成一个 PyCodeObject 对象后再一条条执行字节码指令。

虚拟机中的 PyCodeObject 对象什么时候创建？

PyCodeObject 对象的创建时机是模块加载的时候，即 import。

1、 执行 python test.py 会对 test.py 进行编译成字节码并解释执行，但不会生成 test.pyc

2、 如果 test.py 中加载了其他模块，如 import urllib2，那么 python 会对 urllib2.py 进行编译成字节码，生成 urllib2.pyc，然后对字节码解释执行。

3、 如果想生成 test.pyc，我们可以使用 python 内置模块 py_compile 来编译。

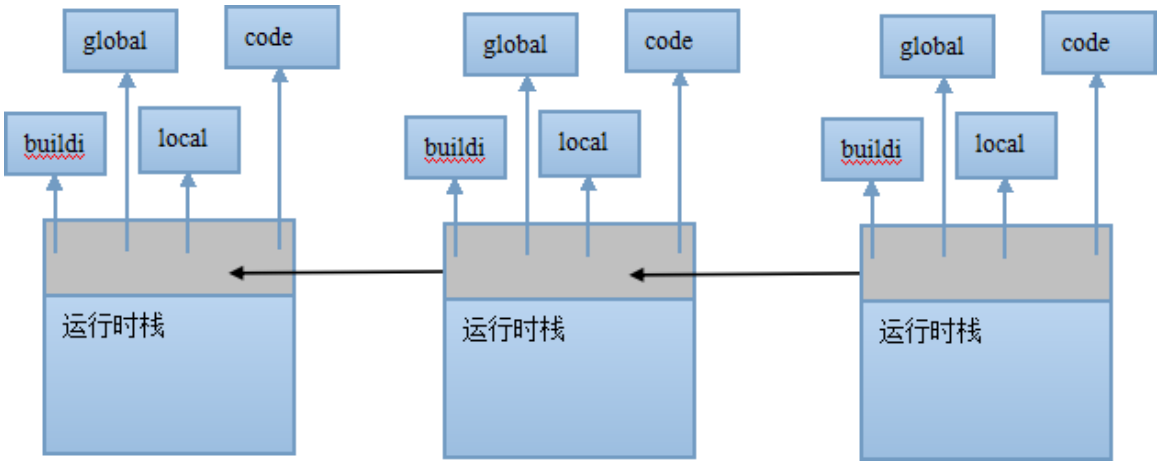
也可以执行命令 python -m test.py 这样，就生成了 test.pyc

4、 加载模块时，如果同时存在.py 和.pyc，python 会使用.pyc 运行，如果.pyc 的编译时间早于.py 的时间，则重新编译.py，并更新.pyc 文件。

- **PyFrameObject**

当发生函数调用时，创建新的栈帧，对应 Python 的实现就是 PyFrameObject 对象。

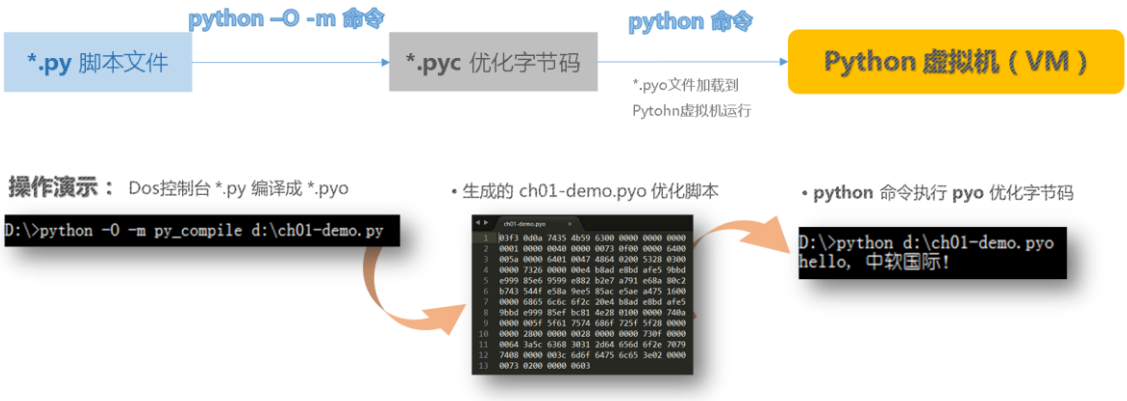
那么对应 Python 的运行时栈就是这样子：



pyo 优化字节码

pyo 是优化编译后的程序 python -O -m 源文件即可将源程序编译为 pyo 文件

特别强调：从 Python3 开始没有 pyo 为后缀名的文件，取而代之是生成对应的 pyc 文件



pyd 是 python 的动态链接库文件，在这里我们先不做过多的介绍。

■ py、pyc、pyo 和 pyd 之间的区别

py 是源文件，pyc 是源文件编译后的字节码文件，pyo 是源文件优化编译后的字节码文件，pyd 是其他语言写的 Python 库

Python 并非完全是解释性语言，它是有编译的，先把源码 py 文件编译成 pyc 或者 pyo，然后由 python 的虚拟机执行，相对于 py 文件来说，编译成 pyc 和 pyo 本质上和 py 没有太大区别，只是对于这个模块的加载速度提高了，并没有提高代码的执行速度。

注意：官方文档上说脚本代码中只要使用引用模块 import 模块库 ，那么 模块库.py 就会先编译成 pyc 然后加载运行。

1.3 Python 在 Windows 系统中的开发环境搭建

1.3.1 Python版本介绍

Python 语言在长期的发展过程中有两个比较常见的版本 Python2.x 和 Python3.x

➢ Python 2.x

该版本为企业应用一个长期版本，也是相对比较稳定的一个版本。目前几乎所有的 Python 项目都在使用该版本。目前主要使用的是 Python2.7.x 这个版本。

➢ Python 3.x （目前的主流为 3.6.x）

相对于早起的 Python 版本，这是一个最新的，也是比较大的一次升级。开发团队为了不带

入过多的累赘，Python 3.0 在设计之初的时候尽量考虑到向下兼容。目前企业很多原与 Python 兼容的工具和程序现在刚刚开始逐步开始进行升级，以适应最新的 Python 版本。

1.3.2 Pip 插件配置

easy_install（已过气） 和 pip

Q：由于 Python 有几乎无限的第三方模块库，那我们如何 安装 和 管理 这些第三方模块呢？

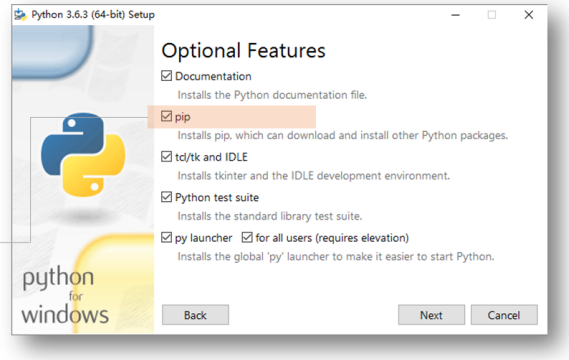
这里我们必须提到 **easy_install** 和 **pip**

老版本中的Python只有 **easy_install** 。

pip 是easy_install的高级版本，

所以 **pip** 和 **easy_install** 任选一个都可以。

而我们目前使用 Python 3.6.5 这个版本，
在安装的时候默认选择了 pip



验证 pip

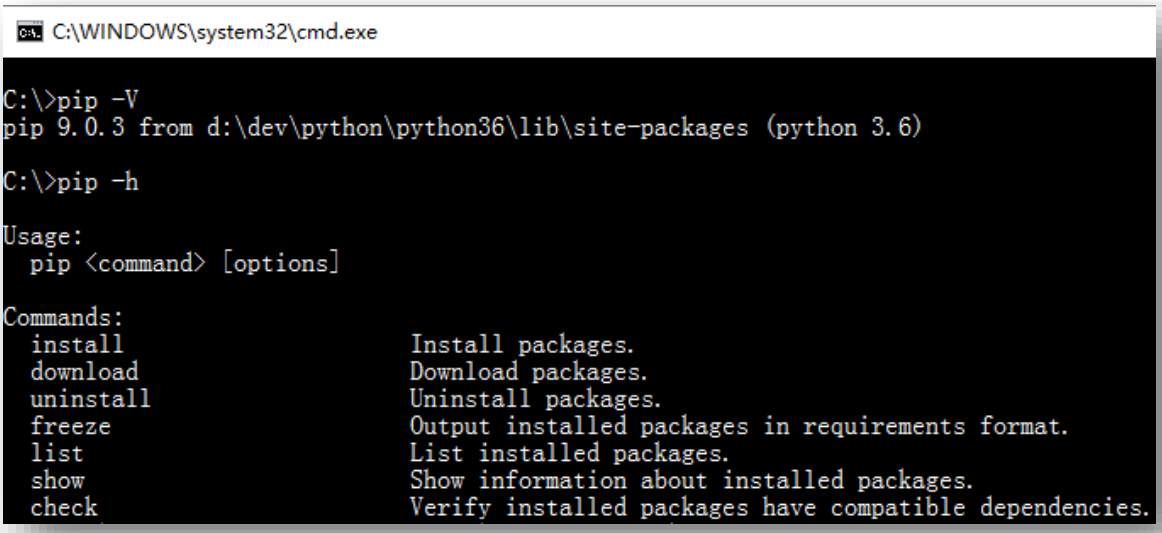
更新完毕 pip 镜像源之后，再进行模块库的下载以及 pip 自身的更新，则全部指向国内镜像源。

Step1：验证 pip 版本号

指令：pip -V

Step2：查看 pip 帮助

指令：python -h



pip 模块管理器的使用（在线安装模块）

pip 默认与 Python 会同时安装到当前系统环境中，其主要的功能就是管理当前系统中的所有 Python 外部模块库。

pip 常用管理指令介绍：

- ✓ pip list：查看当前模块库中已经安装的所有外部模块指令。
- ✓ pip install 模块名称：在线安装外部模块指令（同时会自动下载安装与其相关依赖的模块库）。
- ✓ pip uninstall 模块名称：从模块库中删除指定的模块指令。
- ✓ pip install --upgrade 模块名称：升级指定的模块到最新的版本。

pip 的安装配置

- Python3.6.5 在安装时默认选中的 Pip 模块管理组件。
- setuptools 和 wheel 两个组件都是 Python 安装第三方模块库的依赖工具组件。
- 目前 Python3.6.5 默认安装的 pip 和 setuptools 均为最新版本，但 wheel 模块需手动安装。
- 建议安装 Python 的外部管理模块 wheel：

安装 wheel 指令：pip install -U wheel

pip 模块下载安装的磁盘位置

Python36 安装路径\Lib\site-packages 文件夹中

配置 pip

因为 pip 的服务器一般安装在国外，基于国内糟糕的网络环境，使得 pip 安装 Python 第三方模块将是一个 很痛苦 的过程。

Step1：指定国内 pip 的镜像源

根据 pip 的指南，Windows 中的 pip 的配置文件是 %HOME%/pip/pip.ini（具体到当前环境就是 Windows10 系统的当前用户目录，所以位置是 C:\用户\用户名\pip\pip.ini）。

国内镜像地址：

index-url = https://pypi.tuna.tsinghua.edu.cn/simple

注意：默认情况下 pip 文件夹 和 pip.ini 文件都未被创建，需要自行创建修改；上图中准备了 3 个 pip 源，任选其一都可以。

选择的方法就是在不需要的地址前面加上 # 符号。

扩展：pip 模块管理器国内镜像设置（清华大学镜像源）

- 临时使用

  pip install -i https://pypi.tuna.tsinghua.edu.cn/simple 包名称

  注意，simple 不能少，是 https 而不是 http

- 永久更改

  Linux： ~/.config/pip/pip.conf

  Windows10： C:\用户\<用户名>\pip\pip.ini (pip 文件夹及 pip.ini 需要手动创建)

  macOS： $HOME/Library/Application Support/pip/pip.conf (没有就创建一个)

  修改 index-url 至 tuna，例如

[global]

index-url = https://pypi.tuna.tsinghua.edu.cn/simple

参考：pip 和 pip3 并存时，只需修改 ~/.pip/pip.conf。

1.3.3 Python虚拟环境

virtualenv 虚拟环境（多版本 Python 共存）

- 我们可以在系统中安装多个版本的 Python，为了方便 Python 版本之间的相互切换，我们可以使用 virtualenv（虚拟环境）实现同一系统中多版本共存使用的问题。
- Virtualenv 的安装和配置只需简单 5 步即可完成：

安装前提：系统中已经存在某一个版本的 Python 开发环境（当前为 Python3.6）

Step1：使用 pip install -U virtualenv 下载安装虚拟模块包

Step2：安装其他版本的 Python 开发环境（安装 Python2.7）

Step3：创建虚拟环境文件夹 virtualenv -p *其他版本 pyhton.exe 的路径 虚拟文件夹名称*

Step4：进入当前虚拟环境 cd 虚拟文件夹名称

Step5：启动虚拟环境 虚拟环境文件夹/Script/activate

之后我们就可以使用 pip 等指令在当前环境中安装各种工具模块库

若要停止当前虚拟环境需要在执行 Script 文件中的 deactivate 指令即可。

1.3.4 pywin32的安装配置

- Python 一般基于 Linux 操作系统运行最为常见，若在 windows 系统下使用 Python 则需要安装 pywin32，以便于 Python 全面支持 windows 系统底层编程 api 接口操作。
- pywin32 的安装和配置（系统默认 Python 环境）：

Step1：从网站 https://sourceforge.net/projects/pywin32/files/pywin32/Build%20221 下载对应 python 版本的 exe 安装程序。

Step2：双击安装即可，在安装过程中会自动找到 Python 默认安装路径进行安装。

- Pywin32 在 python 虚拟环境中使用指令安装和配置（个人推荐）：

首先激活虚拟环境使用 activate 指令

执行：pip install pypiwin32 安装

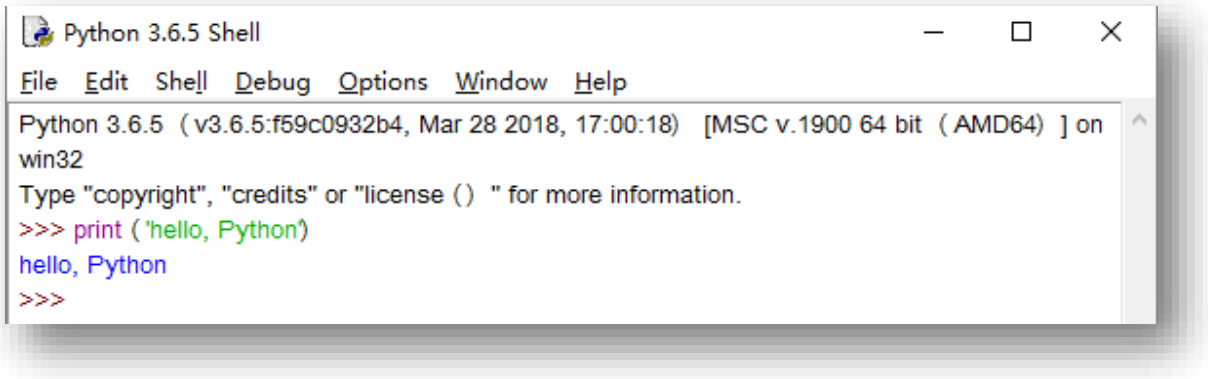
1.3.5 Python 开发IDE介绍

Python 开发工具

■ IDEL

是 Python 软件包自带的集成开发环境，可以方便的创建、运行和调试 python 程序。

- 启动 IDLE 后先看到的是 Python shell,可以通过它在 IDLE 内部执行 python 命令。
- IDLE 还带有一个编辑器，用来编辑 python 程序（或脚本）；
- 有一个交互式解释器用来解释执行 Python 语句；
- 有一个调试器来调试 Python 脚本。

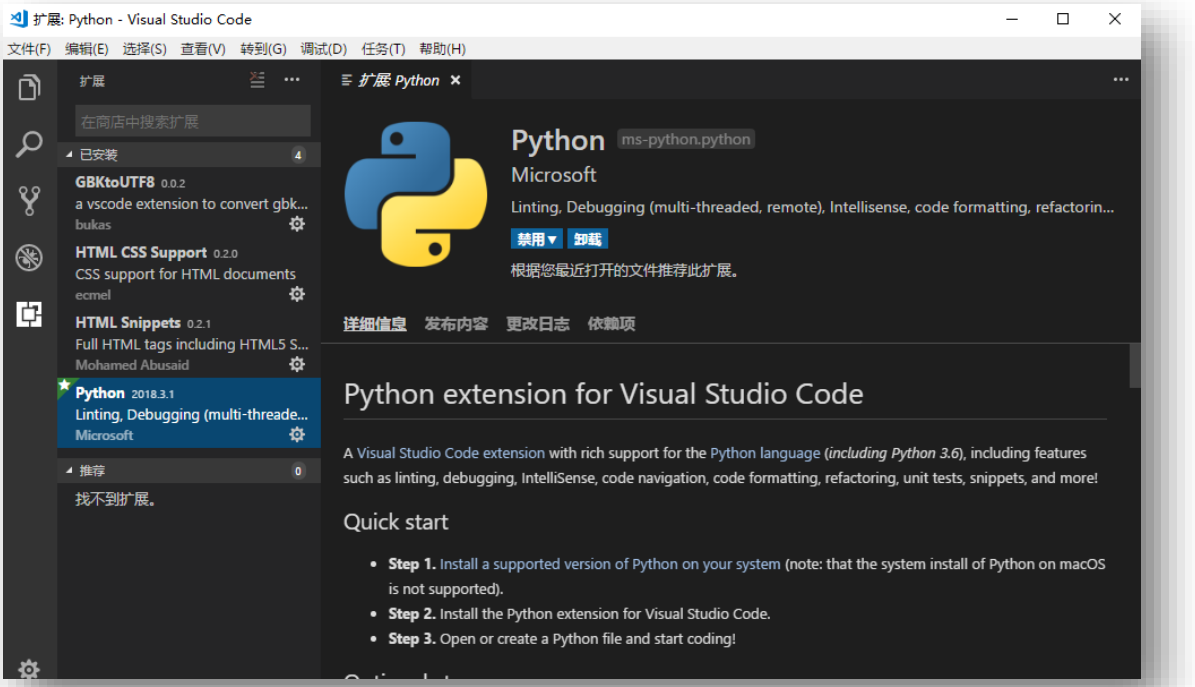


注意：启动 从 windows 的 开始 -> 程序 -> IDEL（Python GUI）

■ VSCode

是微软官方推出的强大的语言编译器，被 Python 开发者广泛使用。

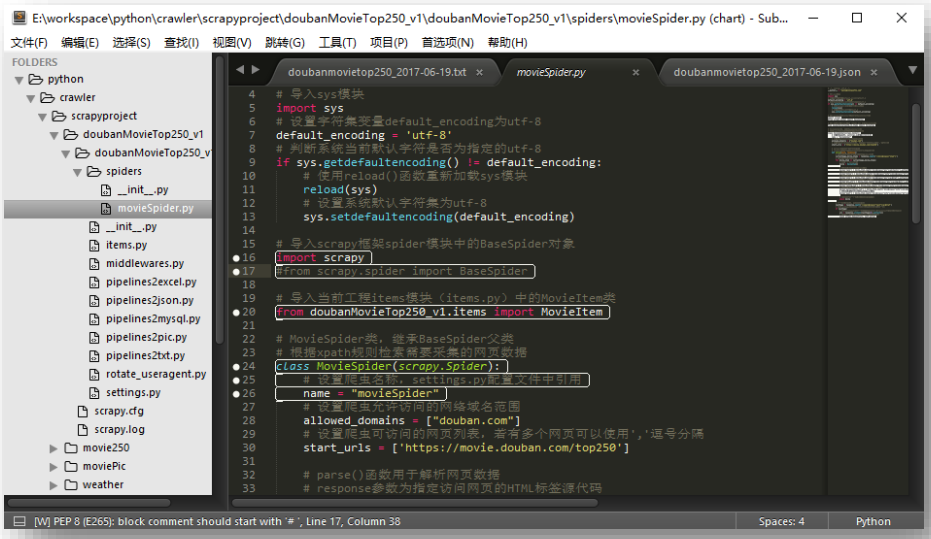
- 强大的插件机制，可将其变身为各种语言的编译器；
- 灵活易用，提示友好；
- Python 开发者首选 IDE。



■ Sublime Text3

是一个代码编辑器，具有漂亮的用户界面和强大的功能。

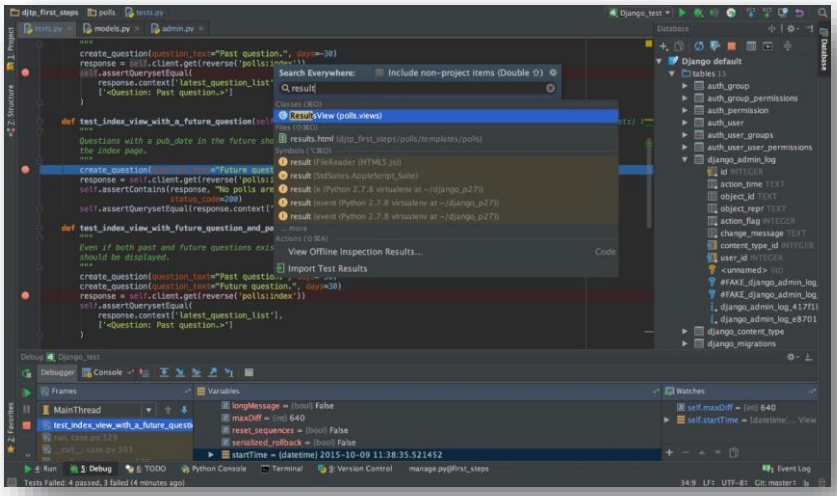
- 强大的插件机制，可将其变身为各种语言的编译器；
- Anaconda 插件篇快速配置 Python 开发环境；
- 运行效率高，被 Python 开发者誉为”Python 利器“。



■ PyCharm

是一种 Python IDE，带有一整套可以帮助用户在使用 Python 语言开发时提高其效率的工具。

比如调试、语法高亮、Project 管理、代码跳转、智能提示、自动完成、单元测试、版本控制。此外，该 IDE 提供了一些高级功能，以用于支持 Django 框架下的专业 Web 开发。

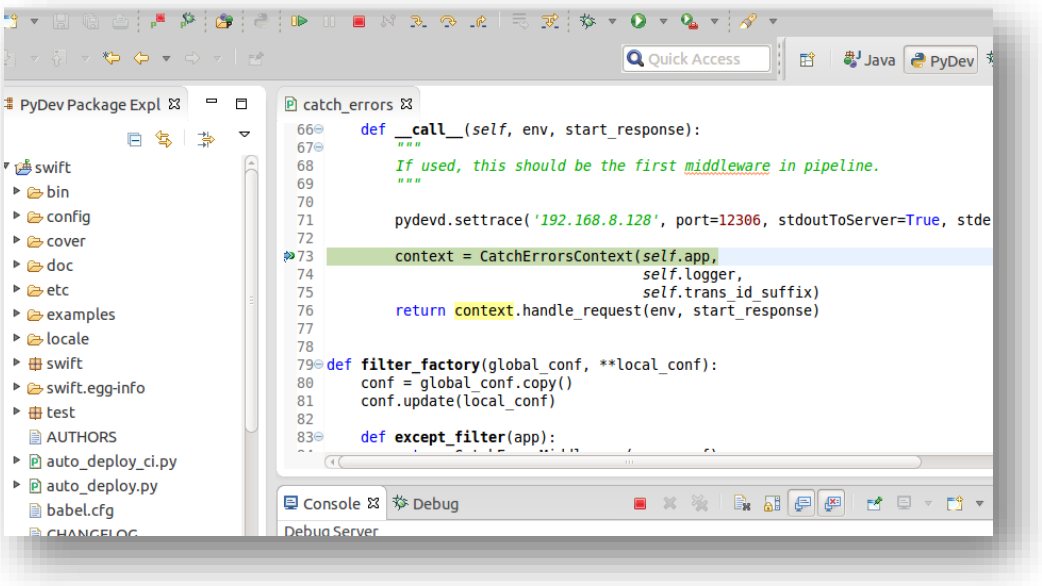


■ PyDev

Eclipse for Python 的一个重要插件，可以将 Eclispe 完全打造成专业的 Python IDE。

PyDev 插件的出现方便了众多的 Python 开发人员，它提供了一些很好的功能，如：语法错误提示、源代码编辑助手、Quick Outline、Globals Browser、Hierarchy View、运行和调试等等。

基于 Eclipse 平台，拥有诸多强大的功能，同时也非常易于使用，PyDev 的这些特性使得它越来越受到人们的关注。



1.3.6 实战任务：VSCode的安装配置

① VSCode 快速安装

② VScode 配置 Python 开发插件

③ VSCode 快速使用入门

VScode 安装配置

VSCode 是微软出的一款轻量级代码编辑器，免费而且功能强大，对各类编程语言以及 JavaScript 和 NodeJS 的支持非常好，自带很多功能，例如代码格式化，代码智能提示补全、Emmet 插件等。VSCode 推荐一个项目以文件夹的方式打开。

目前也是 Python 开发工程师首选 IDE。（免费）

VSCode 开发 IDE 的安装配置

从官方网站 https://code.visualstudio.com/Download 下载对应版本的 VSCode 安装包。

Python 的安装和配置只需简单 3 步即可完成：

Step1：安装 vscode 开发工具

Step2：安装 python 开发插件，常用的如下所示：

python、 *Python-autopep8* 、 *GBKToUTF8* 、 *HTML Snippets*、 *HTML CSS Support*

Step3：配置当前开发工具所使用的虚拟运行环境（不配置则默认为当前系统的 python 默认环境）

在 文件->首选项->设置->工作区设置 ，修改 settings.json 配置文件中添加虚拟环境的路径位置选项即可，则当前工程文件夹将使用虚拟环境的 Python 进行编译执行。如下所示：

“python.pythonPath”: ”D:\\dev\\python\\venvs\\penv36\\Scripts\\python.exe”

3.6 实战任务 2：第一个 Python 程序

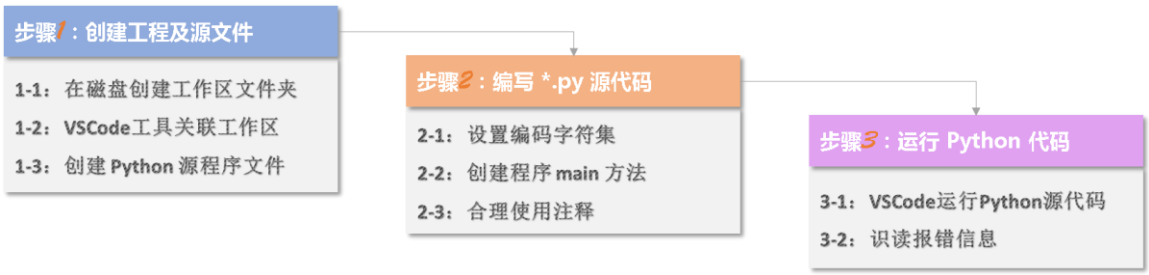
① Python 工程创建标准步骤

② 编写 Python 程序

③ 运行 Python 程序

④ 中文字符处理

3.6.1 Python 开发流程



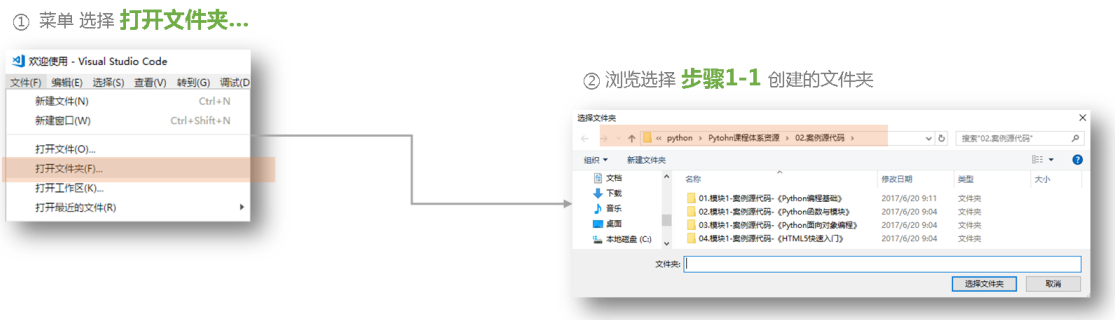
Step1-1：在磁盘创建 Python 工作区文件夹

创建工作区文件夹是一个良好的编程习惯，它可以有效地管理本地代码。

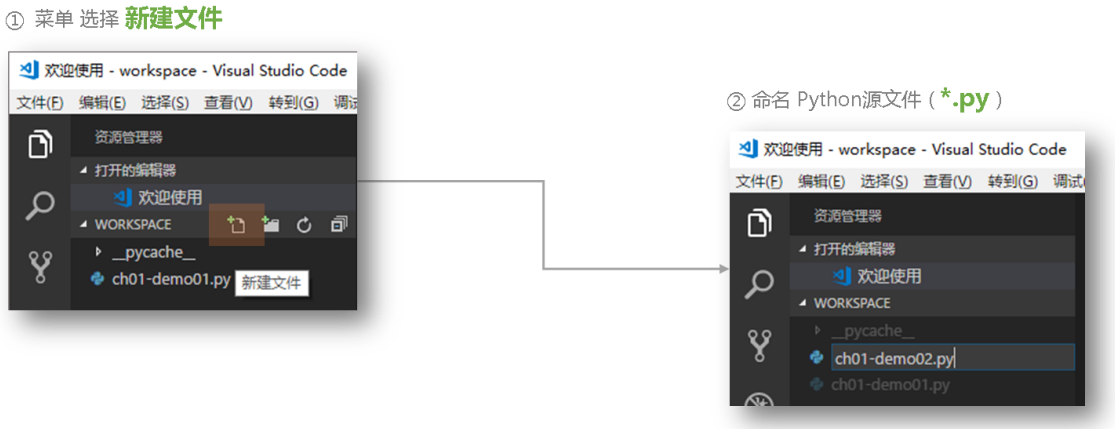
在本地磁盘创建一个文件夹，名称为：workspace

例如：d:\……\01.模块 1-案例源代码-《Python 编程语言》\第 01 章-《Python 快速入门》

Step1-2：VSCode 开发工具关联工作区



Step1-3：创建 Python 脚本程序文件 *.py



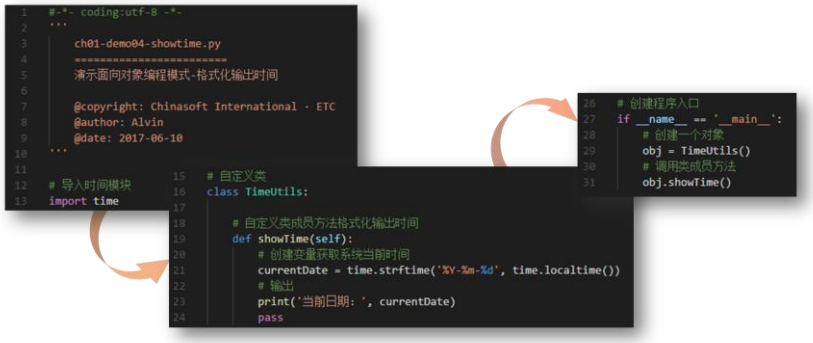
Step2：编写 ch01-demo02-hello.py 源文件（指令编程模式）

PowerShell 无法加载文件 ps1，因为在此系统中禁止执行脚本

步骤 2 编写程序

扩展 1：编写 ch01-demo03-greeting.py 源文件（函数编程模式）

扩展 2：编写 ch01-demo04-showtime.py 源文件（面向对象编程模式）



扩展 3：编写 ch01-demo05-error.py 源文件（查看报错信息）

1.4 本章总结

通过本章的学习，我们初步掌握和了解的 Python 编程语言。对 Python 编程语言的应用领域作用有了初步的认知。通过对 Python 编程语言在系统中的安装配置以及相关组件插件的使用全面掌握了 Python 学习以及前期开发的准备工作。

第02章：Python语法规范与数据类型

知识点

目标 1：了解 Python 的两种编程模式

目标 2：掌握 Python 的基本语法

目标 3：掌握输出及命令行参数的应用

目标 4：掌握 Python 的数据类型以及相关操作

技能点

实战任务 1：使用 Python 命令行参数实现用户模拟注册

2.1 Python 基础语法

2.1.1 Python 编程模式

- **Python 的编程模式**

Python 语言是一个典型的脚本语言，因此它的编程模式（方式）很灵活。常见的有**两种**：

- 交互式命令编程模式
- 脚本编程模式

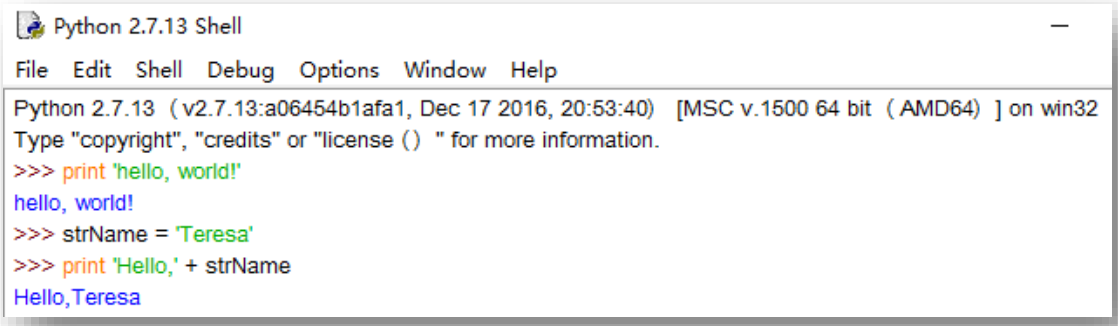
- **交互式命令编程模式**

Python 脚本文件是一种典型的解释型语言，因此其执行的方式为逐行读取逐行执行。而交互式命令编程模式就是一种典型逐行读取执行模式。

当程序只有一行或较少的时候，这种编程模式是比较典型的应用方式。

下图使用 Python IDLE 编辑器进行编程，而该编辑器的编程模式就是典型的交互式命令编程模式。

其特点就是，符号>>>就是输入交互命令的提示符，每次输入完毕后回车，该命令就被 Python 解析器执行。



■ **脚本编程模式**

当我们需要编写较为复杂或大段的代码的时候（特别是在使用函数编程或面向对象编程时），这样的命令式编程就显得很不舒服。

因此，Python 提供了脚本编程模式。可以创建一个后缀名为 *.py 的脚本文件，将大量的代码编写到该文件中，这样便于代码的维护和更新，之后再使用交互命令执行或 IDE 工具运行即可。

```python
#-*- coding:utf-8 -*-
'''
    ch01-homework01.py
    ====================
    作业1：接受用户输入的姓名并输出

    @copyright: Chinasoft International · ETC
    @author: Alvin
    @date: 2018-04-08 19:50
'''

# 脚本程序入口
if __name__ == '__main__':
    # 接受控制台用户输入
    name = input('请输入您的姓名:> ')
    # 输出
    print('你好，%s' % name)
```

2.1.2 字符编码

字符串是一种数据类型。但是，字符串还有一个比较特殊的编码问题。

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。

补充：字符编码发展史

最早的计算机在设计时采用 8 个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是 255（二进制 11111111=十进制 255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是 65535，4 个字节可以表示的最大整数是 4294967295。

由于计算机是美国人发明的，因此，最早只有 127 个字符被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为 **ASCII 编码**，比如大写字母 A 的编码是 65，小写字母 z 的编码是 122。

ASCII 码		字符	ASCII 码		字符	ASCII 码		字符	ASCII		
十进位	十六进位		十进位	十六进位		十进位	十六进位		十进位	十六进	
032	20		056	38	8	080	50	P	104	68	
033	21	!	057	39	9	081	51	Q	105	69	
034	22	"	058	3A	:	082	52	R	106	6A	j
035	23	#	059	3B	;	083	53	S	107	6B	k
036	24	$	060	3C	<	084	54	T	108	6C	l
037	25	%	061	3D	=	085	55	U	109	6D	m
038	26	&	062	3E	>	086	56	V	110	6E	n
039	27	'	063	3F	?	087	57	W	111	6F	o
040	28	(	064	40	@	088	58	X	112	70	p
041	29	)	065	41	A	089	59	Y	113	71	q
042	2A	*	066	42	B	090	5A	Z	114	72	r
043	2B	+	067	43	C	091	5B	[	115	73	s
044	2C	,	068	44	D	092	5C	\	116	74	t
045	2D	-	069	45	E	093	5D	]	117	75	u
046	2E	.	070	46	F	094	5E	^	118	76	v
047	2F	/	071	47	G	095	5F	_	119	77	w
048	30	0	072	48	H	096	60	`	120	78	x
049	31	1	073	49	I	097	61	a	121	79	y
050	32	2	074	4A	J	098	62	b	122	7A	z
051	33	3	075	4B	K	099	63	c	123	7B	{
52	34	4	076	4C	L	100	64	d	124	7C	|
	35	5	077	4D	M	101	65	e	125	7D	}
	36	6	078	4E	N	102	66	f	126	7E	~
	7		079	4F	O	103	67	g	127	7F	

Python 字符编码

扩展：Unicode 字符集

- Python3 之所以能够很好地解决中文乱码问题，在于其将所有的字符串都是用 unicode 进行字符编码。
- Unicode 把所有的语言统一到一套编码里，这样就不会有乱码了。
- Unicode 也在不断的发展，但最常用的是用两个字节表示一个字符（如果遇到非常生僻的字符，就需要 4 个字节）。现在我们见到的大多数操作系统和大多数编程语言都支持 unicode。

ASCII 编码和 Unicode 编码的区别

ASCII 编码是 1 个字节，而 Unicode 编码通常是 2 个字节

扩展：UTF-8 字符集

新的问题又出现了：如果统一成 Unicode 编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上就十分不划算。

解决办法的诞生：又出现了把 Unicode 编码转化为“可变长编码”的 UTF-8 编码。

- UTF-8 编码把一个 Unicode 字符根据不同的数字大小编码成 1-6 个字节，常用的英文字母被编码成 1 个字节，汉字通常是 3 个字节，只有很生僻的字符才会被编码成 4-6 个字节。
- 如果你要传输的文本包含大量英文字符，用 UTF-8 编码就能节省空间。
- UTF-8 编码有一个额外的好处，就是 ASCII 编码实际上可以被看成是 UTF-8 编码的一部分，所以，大量只支持 ASCII 编码的历史遗留软件可以在 UTF-8 编码下继续工作。

特别注意：计算机内存中，统一使用 Unicode 编码

补充：ASCII、Unicode、UTF-8 三种字符编码的关系

- Python3 之所以能够很好地解决中文乱码问题，在于其将所有的字符串都是用 unicode 进行字符编码。
- Unicode 把所有的语言统一到一套编码里，这样就不会有乱码了。
- Unicode 也在不断的发展，但最常用的是用两个字节表示一个字符（如果遇到非常生僻的字符，就需要 4 个字节）。现在我们见到的大多数操作系统和大多数编程语言都支持 unicode。

ASCII 编码和 Unicode 编码的区别

ASCII 编码是 1 个字节，而 Unicode 编码通常是 2 个字节

Python3 字符编码

在 Python3 版本中，字符串都是以 Unicode 编码的，也就是说，Python 字符串支持多语言。

单个字符的编码，Python 提供了 **ord（）**函数获取单个字符的十进制整数表示，**chr（）**函数把编码转换成对应的字符。

示例：

```
>>> ord(‘A’)
65
>>> ord(‘中’)
20013
>>> chr(66)
‘B’
>>> chr(25991)
‘文’
```

Python 源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为 UTF-8 编码。当 Python 解释器读取源代码时，为了让它按 UTF-8 编码读取，我们通常在文件开头写上这行

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

注释是为了告诉 Python 解释器，按照 UTF-8 编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

2.1.3 标识符和保留字

1） Python 标识符

2） Python 保留字

■ **什么是标识符？**

标识符(Identifier)是指用来标识某个实体的一个符号。在不同的应用环境下有不同的含义。

举例说明：

- 在日常生活中，标示符是用来指定某个东西、人，要用到它，他或她的名字；
- 在数学中解方程时，我们也常常用到这样或那样的变量名或函数名；

在编程语言中，标识符是用户编程时使用的名字，对于*变量*、*常量*、*函数*、*语句块* 也有名字，统称之为标识符。

■ 标识符的命名规则

在编程语言中，标识符就是程序员自己规定的具有特定含义的词，比如类名称，属性名称，变量名等。

- 在 Python 里，标识符有**字母**、**数字**、**下划线**组成，但**不能以数字开头**。
- Python 中的标识符是**区分大小写**的。
- 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 *** 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

代码演示：

```
>>> print ('你好，中软国际') ; print('hello,chinasoft!')
你好，中软国际
hello,chinasoft!
```

但在通常情况下，我们的 Python 语句无需分号 ； 结束

示例代码：

```
print('你好，中软国际')
print('hello,chinasoft!')
```

■ 什么是保留字？

保留字(Reserved Word)，指在高级语言中已经定义过的字，使用者不能再将这些字作为变量、常量、函数、语句块 等的命名使用。

- 保留字包括关键字和未使用的保留字。
- 关键字则指在语言中有特定含义（如 for / if / pass / ……），成为语法中一部分的那些字。

Python 中的保留字

下面的列表显示了在 Python 中的保留字。这些保留字不能用作常数或变数，或任何其他标识符名称。所有 Python 的关键字只包含**小写字母**。

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	elif
import	try	....

随着我们对 Python 的深入学习，我们会了解更多的保留字。

2.1.4 语法格式规范

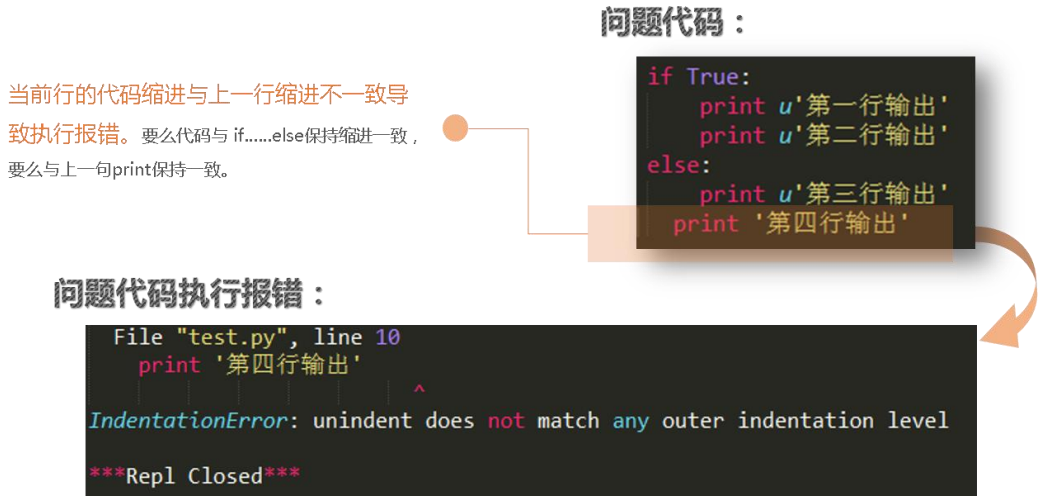
1) 行和缩进
2) 多行语句
3) Python 引号
4) Python 空行

■ **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 { } 来控制区分 函数、逻辑判断和代码块 等语句块的作用域范围和控制区域。python 最具特色的就是用缩进来写模块。

缩进的空白数量是可变的（一般是 1 个 Tab 制表位），但是所有代码块语句必须包含相同的缩进空白数量，具有行缩进一致的相邻代码被认定为是 1 个块结构，这个必须严格执行。如下所示：

我们来看一段初学者经常会犯错误的代码：



我们有几种方法可以改正以上代码让其正确运行呢？

由于行缩进导致编码执行报错，经常会出现以下两种情况，我们分别说明一下：

IndentationError: unexpected indent 错误是文件里格式不对，可能是 tab 和空格没对齐的问题。所有 python 对格式要求非常严格。

IndentationError: unindent does not match any outer indentation level 错误表明使用的缩进方式不一致，有的是 tab 键缩进，有的是空格缩进，改为一致即可。

因此，在 Python 的代码块中必须使用相同数目的行首缩进空格数。

建议：在每个缩进层次使用 单个制表符 或 两个空格 或 四个空格 ，切记不能混用

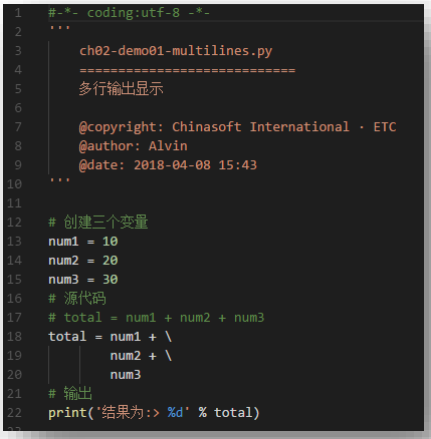
■ 编码多行显示

Python 语句中一般以新行作为为语句的结束符。

有的时候一行代码太长，不便于我们书写清晰的代码结构，还可能造成代码阅读起来很不方便。

因此，我们可以使用斜杠（ \）将一行的语句分为多行显示，如下所示：

代码演示 ch02-demo01-multilines.py：



```python
#-*- coding:utf-8 -*-
'''
    ch02-demo01-multilines.py
    ============================
    多行输出显示

    @copyright: Chinasoft International · ETC
    @author: Alvin
    @date: 2018-04-08 15:43
'''

# 创建三个变量
num1 = 10
num2 = 20
num3 = 30
# 源代码
# total = num1 + num2 + num3
total = num1 + \
        num2 + \
        num3
# 输出
print('结果为:> %d' % total)
```

这段代码的［第 18~20 行］语句使用斜杠（ \） 将多行代码链接，运行结果正常 >>>**总和：60**

■ 多行语句

语句中包含 [ ], { } 或 ( ) 括号就不需要使用多行连接符，因为它们属于序列数据类型。

如下实例：

代码演示 ch02-demo02-sequence.py ：

```python
#-*- coding:utf-8 -*-
'''
    ch02-demo02-sequence.py
    ============================
    序列对象的对行显示

    @copyright: Chinasoft International · ETC
    @author: Alvin
    @date: 2018-04-08 15:43
'''
# 源代码
# persons = ['张三丰', '张翠山', '张无忌',]
# 创建一个序列对象-列表（多行）
persons = ['张三丰',
           '张翠山',
           '张无忌',]
# 输出
print('人数总计:> %d' % len(persons))
```

这段代码的［第 14~16 行］语句无需使用斜杠（ \） ，运行结果正常 >>>**人数为：3**

2.2 Python 注释及规范

2.2.1 注释及其他

① Python 注释

② 标准输入 input()

③ 标准输出 print( )输出

④ 命令行参数

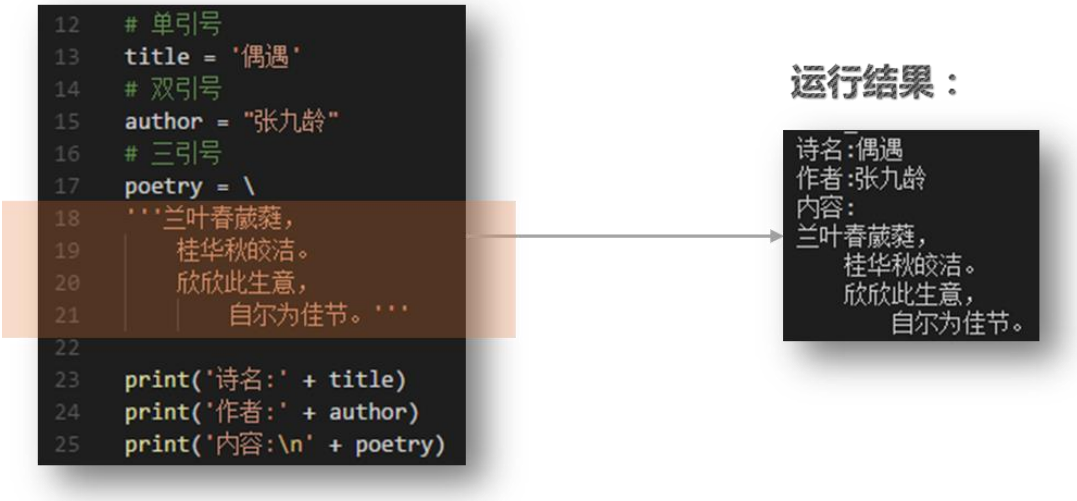
2.2.2 引号使用说明

Python 可以使用单引号(’)、双引号(″)、三引号(’’’ 或 ″″″) 来表示字符串

① 引号的开始与结束必须的相同类型的；

② 其中三引号可以由多行组成，编写多行文本的快捷语法，常用于文档字符串，在文件的特定地点，被当做注释。

代码演示 ch02-demo03-quotes.py ：



2.2.3 Python 注释

在编程语言中，注释的作用是为了让自己或他人更快地了解程序作者的思路和意图，提高代码的可读性。同时在多人协同开发时，也可以提高开发效率。

特备说明：注释部分不参与代码的编译执行

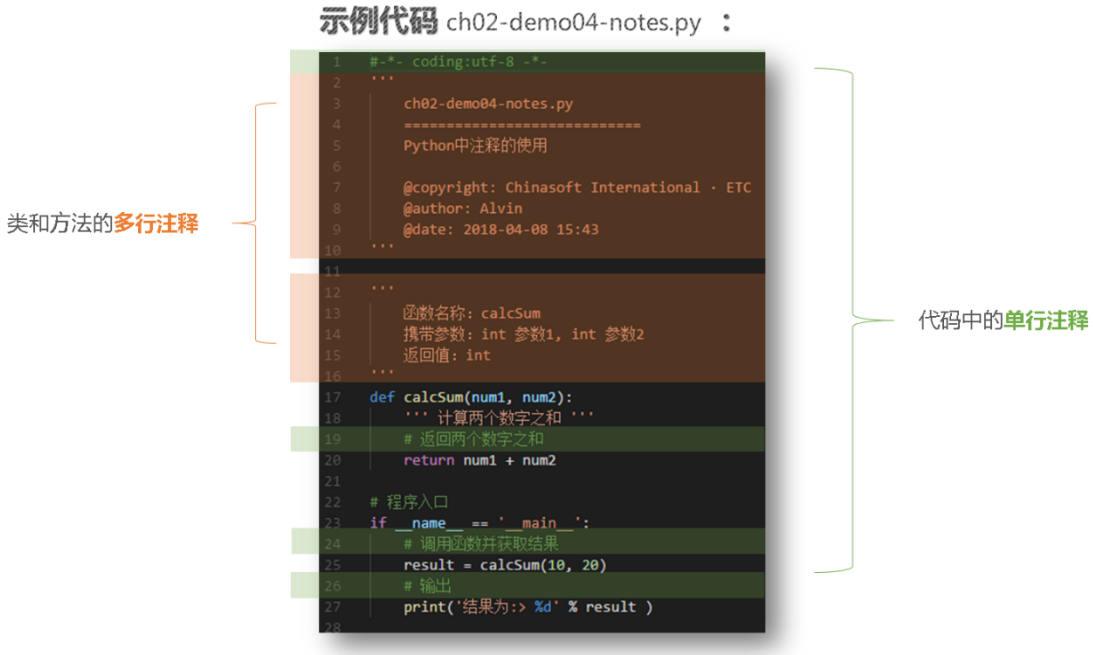
单行注释 #

单行注释主要应用于对某个变量，代码等的简短说明，不能换行，只能在 1 行内应用。

多行注释（三个单引号） ''' 或 （三个双引号） """

多行注释主要应用于大段文字的说明，可以换行使用。一般用于对类/函数的注释（类注释也可以单行）。

我们通过一段典型的代码，了解一下注释在实际开发中的应用，如下所示：



2.2.5 Python 空行

空行是编程过程中，函数之间或类的方法之间实现的**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，从而让**代码结构更加清晰易读**。

- 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。
- 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。

空行也是程序代码的一个组成部分；

空行在 Python 中也可已使用关键字 `pass` 表示

```python
#-*- coding:utf-8 -*-
'''
    ch02-demo04-notes.py
    ============================
    Python中注释的使用

    @copyright: Chinasoft International · ETC
    @author: Alvin
    @date: 2018-04-08 15:43
'''

'''
    函数名称：calcSum
    携带参数：int 参数1, int 参数2
    返回值：int
'''
def calcSum(num1, num2):
    ''' 计算两个数字之和 '''
    # 返回两个数字之和
    return num1 + num2

# 程序入口
if __name__ == '__main__':
    # 调用函数并获取结果
    result = calcSum(10, 20)
    # 输出
    print('结果为:> %d' % result )

```

2.2.5 等待用户输入

等待用户输入，实际上是将当前运行的程序线程挂起，暂停程序的运行。等待用户交互操作之后，在按回车或输入特定字符之后，恢复程序挂起的线程，继续执行，同时处理输入的数据。

- input(‘……*输入提示内容*……’) 该函数用接收接收的数据全部为 str 字符串类型；
- 若想转换成成其他类型，需使用强制类型转换，如 类型名称(str 数据)。

示例代码 ch02-demo05-input.py ：

```python
#-*- coding:utf-8 -*-
'''
    ch02-demo05-input.py
    ========================
    等待用户输入

    @copyright: Chinasoft International · ETC
    @author: Alvin
    @date: 2018-04-08 15:43
'''

# 接收用户输入的姓名
name = input('请输入您的姓名:> ')
print('姓名:> %s' % name)
# 接受用户输入的年龄
age = input('请输入您的年龄:> ')
print('年龄:> %d' % int(age))
```

程序运行

```
请输入您的姓名:> 张三
姓名:> 张三
请输入您的年龄:> 10
年龄:> 10
```

2.2.6 print 屏幕输出

print( ) 打印输出函数是在开发中用得很多的函数，代表输出并换行。其语法结构也有很多……

格式化输出规范 1：

- print(字符常量 + 字符变量)　　**说明：** + 加号仅用于连接两个字符串类型
- print(字符常量 ，任意数据类型)　**说明：**，逗号用于连接任意数据类型
- print(输出的数据 ，end=‘ ’)　　**说明：**输出不换行，与下一个输出在同行显示，并使用 end 指定的字符连接

示例代码 ch02-demo06-output.py ：

```
12  # 声明一个字符串类型数据
13  name = 'Teresa'
14  # 声明一个数字类型数据
15  age = 12
16
17  # 输出方式1
18  print('姓名: ' + name)
19  print('年龄: ' , age)
20  # 同行输出
21  print('姓名: ' + name, end=' ')
22  print('年龄: ', age)
```

print（字符串常量 + 字符串变量）

print（字符串常量 ， 数字类型）

输出不换行

除了简单的数据连接输出之外，Python 语言同样支持多种形式的占位符格式化输出模式。

格式化输出规范 2：

- print(格式占位符号 % 变量名称)
- print(多个格式占位符号 %(变量 1, 变量 2, …. , 变量 N))
-

常用格式占位符号 ：%s：输出字符串类型； **%d**：整数类型；**%f**：浮点数类型（小数） 等等.

- **%-10s**：占位 10 个字符，左对齐，多与的占位使用空格填充
- **%-8.2f**：占位 8 个字符，左对齐，**.2** 代表小数点保留两位

示例代码 ch02-demo06-output.py

```
24  # 输出方式2:
25  height = 1.68 # 设置一个身高浮点类型数据
26  print('姓名: %s, 年龄: %d, 身高: %f' %(name, age, height))
27  print('姓名: %10s, 年龄: %-10d, 身高: %.1f' %(name, age, height))
```

```
姓名: Teresa, 年龄: 12, 身高: 1.680000
姓名:     Teresa, 年龄: 12        , 身高: 1.7
```

同样，Python 还支持一种 **format** 格式输出模式。

格式化输出规范 3：

print(‘{0} + {1} = {2}’ **.format**(num1, num2, num1 + num2))

说明：｛ ｝方式为占位的另一种表现，但后续需要通过使用 format 函数绑定变量

变量或数据的个数要与占位符的个数保持一致。

示例代码 ch02-demo06-output.py ：

```
29  # 输出方式3
30  print('姓名: {0}'.format(name))
31  print('年龄: {0}, 身高: {1}'.format(age, height))
```

```
姓名: Teresa
年龄: 12, 身高: 1.68
```

2.3 命令行参数

2.3.1 什么是命令行参数

在执行 Python 命令的时候需要携带的参数称之为 命令行参数。

举例说明

C:\ dir e:　　dir 是 Dos 命令，用于查看指定磁盘位置目录，e: 是 该命令的参数，确定查看 E 盘目录

示例代码：

>>>python -m py_compile d:\demo01.py　　*-m py_compile d:\demo01.py* 就是 python 的命令行参数

- **Python 虚拟机在接收到命令之后自动执行对应的脚本**

#导入 py_compile 模块

import py_compile

#使用解析器将*.py 编译成*.pyc，path 代表*.py 的文件路径 py_compile.compile(‘path’)

Python 中也可以所用 **sys** 内置模块的 **sys.argv** 来获取命令行参数：

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

示例代码 ch02-demo07-argv.py ：

```python
# 导入sys内置模块
import sys

# 使用len()函数获取命令行参数的个数
print('命令行参数的个数为 %d 个参数' % (len(sys.argv)))
# 输出命令行参数名称
print('命令行参数的名称为: ', sys.argv)
# 输出命令行第2个参数的名称和类型
arg2 = sys.argv[1]
print('命令行第2个参数的名称: {0}, 类型为: {1}'.format(arg2, type(arg2)))
```

注意：所有参数无论什么类型都被程序转换成字符串类型

控制台向程序传递命令行参数 (多个参数使用空格分隔)

```
(penv36) PS D:\dev\python\workspace\M1_Course01_Demos> cd .\CH02_Demos\
(penv36) PS D:\dev\python\workspace\M1_Course01_Demos\CH02_Demos> python .\ch02-demo07-argvs.py 'firstArgument'
命令行参数的个数为 2 个参数
命令行参数的名称为:  ['.\\ch02-demo07-argvs.py', 'firstArgument']
命令行第2个参数的名称: firstArgument, 类型为: <class 'str'>
(penv36) PS D:\dev\python\workspace\M1_Course01_Demos\CH02_Demos>
```

2.3.2 实战任务：模拟用户个人信息注册

使用 Python 命令行参数向脚本程序输入参数并显示

业务需求：

模拟用户个人信息注册，需要输入用户个人信息 姓名、性别、年龄、血型、身高、电话 信息，并输出显示。

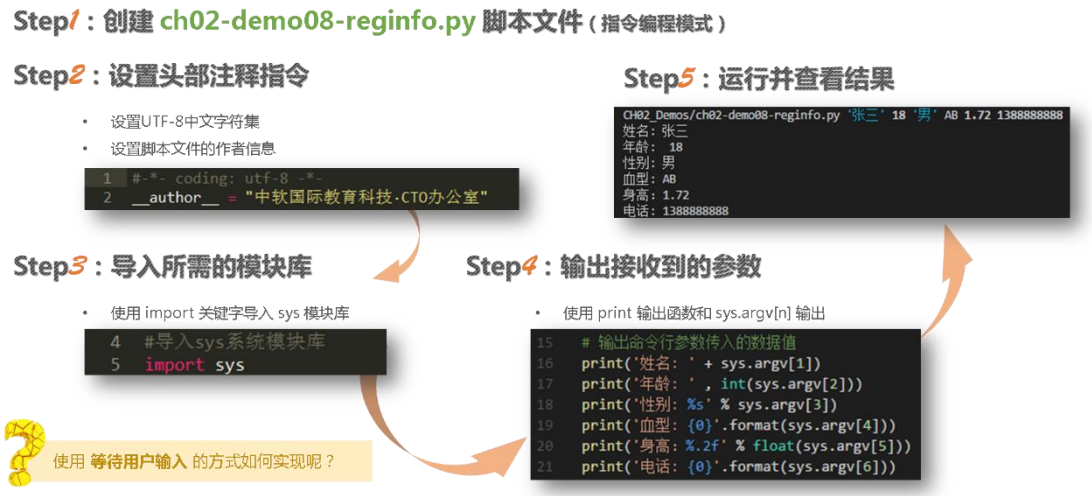
技术需求：

- 使用命令行参数方式向脚本程序传递相关参数；
- 格式化输出显示信息

关键技术分析：

- 使用 sys.argv[参数下标] 输出显示
- 使用占位符及格式输出符输出信息

实现步骤



2.4 基本数据类型

2.4.1 变量基本概念

介绍什么是变量，以及变量的基本使用方法。通过对内存结构的分析，让大家对变量有更层次的认知。

2.4.2 什么是变量？

变量 来源于数学，是计算机语言中能 储存计算结果 或 能表示值 的一个抽象概念（可以理解为一个代号）。

变量的特点：

- 变量可以通过变量名访问；
- 在指令式语言中，变量通常是可变的。

变量命名的规范：

- 与之前介绍的中 **标识符** 的命名规范是一致的，因为变量名就是一个非常典型的标识符。

2.4.3 数据类型

什么是数据类型？

- 既然变量可以存放各种数据，这就意味着在创建变量时会在内存中开辟一个空间。
- 由于所存储的数据类型各异（如，姓名->字符类型 / 年龄->数字类型），Python 虚拟

机就会决定为不同的类型数据开辟大小不同的内存空间。

因此，变量可以指定不同的数据类型，这些变量可以存储 **整数**，**小数** 或 **字符**。

变量在内存中的表现：

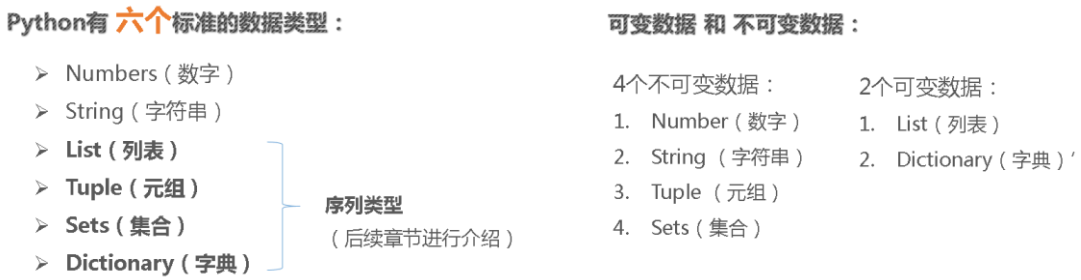
2.4.4 标准数据类型

在内存中存储的数据可以有多种类型。

举例说明：

一个人的年龄可以用数字来存储，他的名字可以用字符来存储

Python 定义了一些标准类型，用于存储各种类型的数据。



2.4.5 变量赋值

变量赋值说明：

- Python 中的变量赋值不需要类型声明；
- 每个变量在内存中创建，都包括变量的标识，名称和数据这些信息；
- 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。

变量赋值运算符：

- 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；
- 赋值语法： **变量名 = 值**

示例代码：

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

■ **多个变量赋值**

示例代码：

```
>>> a = b = c = 1
```

Code 代码说明：以上实例，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。

内存表现形式：

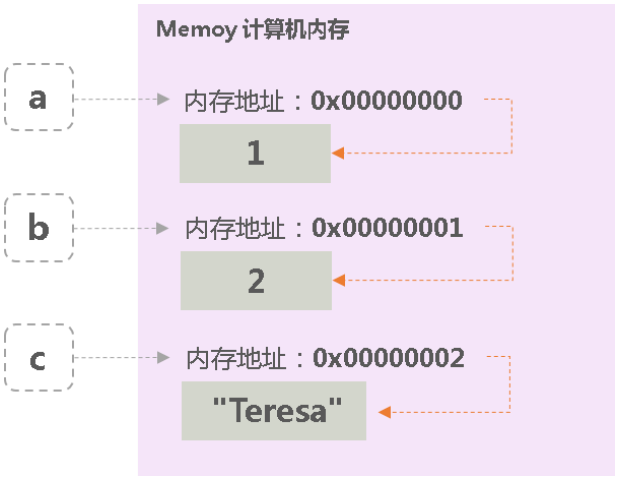
也可以为多个对象指定多个变量。

示例代码：

```
>>> a, b, c = 1, 2, "Teresa"
```

Code 代码说明：两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象 "Teresa" 分配给变量 c。

内存表现形式：



2.4.6 标准数据类型

本小节介绍 数字、字符串。对于列表、元组、集合、字典后续专门章节详细介绍各类型数据的具体操作。

■ **数字类型**

- **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。

示例代码：当你指定一个值时，Number 对象就会被创建

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

- 当你删除一个值时
- Del 语句的语法是：**del var1[, var2[, var3[…….. , varN]]]**

示例代码：当你指定一个值时，Number 对象就会被创建

```
>>> del var
>>> del var_a, var_b
```

Python 支持 四种 不同的数字类型：

- int（有符号整型，通常被称为是整型或长整数，是正或负整数，不带小数点）
- 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))
- bool（布尔类型，由 True 或 False，即 真 1 或 假 0 python2.7 中。常用于逻辑判断使用）
- 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）

数字类型实例

int	bool	float	complex
10	True / False	0.0	3.14j
100		15.20	45.j
-786		-21.9	9.322e-36j
080		32.3e+18	.876j
-0490		-90.	-.6545+0J
-0x260		-32.54e100	3e+26J
0x69		70.2E-12	4.53e-7j

扩充说明：

- 在 Python 3 里，只有一种整数类型 int，表示为长整型，已经取消 python2 中的 Long。
- 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。

■ **字符串类型**

- 字符串或串(String)是由数字、字母、下划线组成的一串字符。他们是**不可改变**的数据类型。
- 一般情况表示为：

  >>> s = ‘ABCDE……’

  >>> s = “ABCDE……”

它是编程语言中表示文本的数据类型。

Python 的字串列表有 2 种 取值顺序:

- 从左到右下标索引默认 0 开始的，最大范围是(字符串长度-1)
- 从右到左下标索引默认-1 开始的，最大范围是字符串开头

如何访问字符串？

如果你要实现从字符串中获取一段子字符串的话，可以使用变量 **[头下标 : 尾下标：步长]**，就可以截取相应的字符串，其中下标是从 **0 开始算起**，可以是正数或负数，下标可以为空表示取到头或尾。

示例代码：

>>> s = ‘ilovechina’

Code 代码说明：

- **s[1:5]** 的结果是 love。

- 当使用以冒号分隔的字符串，python 返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。
- 上面的结果包含了 s[1] 的值 l，而取到的最大范围不包括上边界，就是 s[5]的值 c。

- 加号（+）是字符串连接运算符
- 星号（*）是重复操作。

代码演示 ch02-demo09-string.py ：

```
# 创建一个字符串对象
s = 'Hello Python!'

# 输出
print('%s' % s)       # 输出完整的字符串
print('%s' % s[0])    # 输出字符串中的第1个字符
print('%s' % s[2:5])  # 输出字符串第3个元素至第5个元素
print('%s' % s[6:])   # 输出从第6个元素开始到结束
print('%s' % s[6::2]) # 输出从第6个元素开始间隔2个元素至结束
print('%s' % s[-1])   # 输出最后一个元素
print(s * 2)          # 输出两次字符串
print(s + 'let`s go') # 字符串连接
```

运行结果：

```
Hello Python!
H
llo
Python!
Pto!
!
Hello Python!Hello Python!
Hello Python!let`s go
```

能否替换字符串 s 中的字符？

■ 字符串转义

由于 Python 中的字符串类型使用单引号或双引号包围，因此在字符串中若包含单引号或双引号输出则需要进行字符转义处理。

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)
\n	换行
\r	回车

■ 字符串运算符 in 和 not in

我们经常可以判断字符串中是否包含某个指定的字符或子字符串，这个时候就需要使用 in 或 not in 来进行操作。

示例代码：

```
>>> s = ‘ilovechina’
>>> ‘c’ in s
True
>>> ‘love’ not in s
False
```

■ **字符串函数**

字符串常用内建函数

函数名称：**count**

语法结构：str.**count**(sub, star=0, end=len(string))

函数描述：count() 方法用于统计字符串里某个字符出现的次数。可选参数为在字符串搜索的开始与结束位置。

参数说明：

- sub -- 搜索的子字符串
- start -- 字符串开始搜索的位置。默认为第一个字符,第一个字符索引值为 0。
- end -- 字符串中结束搜索的位置。字符中第一个字符的索引为 0。默认为字符串的最后一个位置

函数返回值：

- 方法返回子字符串在字符串中出现的次数。

示例代码：

- >>> str = ‘ilovechina’
- >>> print str.count(‘i’) #输出 i 在字符串 str 中出现的次数

函数名称：**endswith**

语法结构：str. **endswith**(suffix[, start[, end]])

函数描述：count() 方法用于统计字符串里某个字符出现的次数。可选参数为在字符串搜索的开始与结束位置。

参数说明：

- suffix -- 该参数可以是一个字符串或者是一个元素。
- start -- 字符串中的开始位置。

- end -- 字符中结束位置。

函数返回值：

- 如果字符串含有指定的后缀返回 True，否则返回 False。

示例代码：

```
>>> str = 'ilovechina'
>>> print str.endswith( 'ina' )   #True
```

函数名称：**find**

语法结构：str. **find**(str, beg=0, end=len(string))

函数描述：检测字符串中是否包含子字符串 str ，如果指定 beg（开始） 和 end（结束） 范围，则检查是否包含在指定范围内，如果包含子字符串返回开始的索引值，否则返回-1。

参数说明：

- str -- 指定检索的字符串
- beg -- 开始索引，默认为 0。
- end -- 结束索引，默认为字符串的长度。

函数返回值：

- 包含子字符串返回开始的索引值，否则返回-1。

示例代码：

```
>>> str = 'ilovechina'
>>> print str.find( 'love' )   #1
```

函数名称：**index**

语法结构：str. **index**(str, beg=0, end=len(string))

函数描述：检测字符串中是否包含子字符串 str ，如果指定 beg（开始） 和 end（结束） 范围，则检查是否包含在指定范围内，该方法与 python find()方法一样，只不过如果 str 不在

string 中会报一个异常。

参数说明：

- str -- 指定检索的字符串
- beg -- 开始索引，默认为 0。
- end -- 结束索引，默认为字符串的长度。

函数返回值：

- 包含子字符串返回开始的索引值，否则抛出异常。

示例代码：

```
>>> str = 'ilovechina'
>>> print str.index( 'love' )   #1
```

函数名称：**replace**

语法结构：str. **replace**(old, new, [, max])

函数描述：把字符串中的 old（旧字符串） 替换成 new(新字符串)，如果指定第三个参数 max，则替换不超过 max 次。

参数说明：

- old -- 将被替换的子字符串。
- new -- 新字符串，用于替换 old 子字符串。
- max -- 可选字符串, 替换不超过 max 次。

函数返回值：

- 返回字符串中的 old（旧字符串） 替换成 new(新字符串)后生成的新字符串，如果指定第三个参数 max，则替换不超过 max 次。

示例代码：

```
>>> str = 'ilovechina'
>>> print str.replace( 'china' , 'Beijing' )   # iloveBeijing
```

函数名称：**split**

语法结构：str. **split**(split [, num=string.count(str)])

函数描述：通过指定分隔符对字符串进行切片，如果参数 num 有指定值，则仅分隔 num 个子

字符串

参数说明：

- str -- 分隔符，默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。
- num -- 分割次数。

函数返回值：

- 分割后的字符串列表。

示例代码：

```
>>> str = 'I love china'
>>> print str.split( ' ' )   # ['i', 'love', 'china']
```

字符串处理函数的汇总分类

- **字符串大小写转换**

  lower() / upper() /swapcase() / title()

- **字符串搜索、替换**

  find() /count() / replace / strip() / lstrip() /rstrip()

- **字符串分割、组合**

  split() / join()

- **字符串编码、解码**

  decode()/encode()

- **字符串测试**

  isalpha() / isdigit() / isspace() / islower() / isupper() / istitle()

2.5 本章总结

语法与基础数据类型

字符编码集合

从 ASCII 编码 -> Unicode 编码 -> UTF-8

如何写出优雅的 Python 代码

标识符、行首缩进、代码注释、空行 pass

标准输入输出

input() / print()

控制台命令行参数处理

sys.argv

基础数据类型

6 个基础数据类型（2 个可变，4 个不可变）

数字数据类型

二进制数据向各种进制之间的转换

字符串数据类型

字符串截取操作、字符串常用函数

第03章：序列类型及运算符

知识点

目标 1：了解序列数据类型

目标 2：掌握列表、元组、字典的基础概念及操作技巧

目标 3：掌握各种运算符的操作规则

实战任务

实战任务 1：使用序列类型完成对数据的临时存储

实战任务 2：SMTP 实现邮件的发送 （补充）

3.1 序列对象（sequence）

“序列”是程序设计中经常用到的数据存储方式。在其他程序设计语言中，“序列”通常被称为”数组“，用于存储相关数据项的数据结构。几乎每一种程序设计语言都提供了“序列”数据结构，如 C 和 Basic 中的一维、多维数组等。

Python 语言中本身并没有数组的概念，但在 Numpy 中提供了数组对象，也弥补的 Python 自身的不足。

3.1.1 序列与数组的区别

- 数组是提供了能够存放同一数据类型且连续的内存空间。
- 序列虽然是连续的存储空间，但可以存放不同数据类型，也可以理解为更加 “高级的数组”。

3.1.2 Python中常用的序列对象

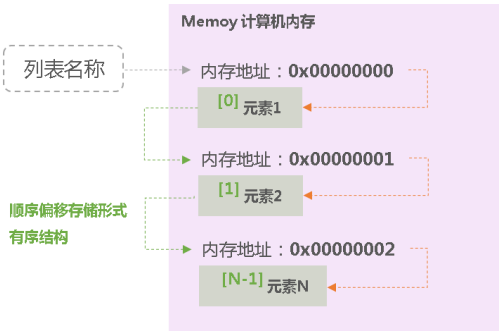
- 列表 List （可变数据类型）
- 元组 Tuple（不可变数据类型）
- 集合 Sets（可变数据类型）
- 字典 Dictionary（可变数据类型）
- 字符串 String（不可变）
- range( )

3.1.3 列表（List）类型

- 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（即嵌套）。
- 列表用" [ ] "标识，是 python 最通用的复合数据类型。

如何创建列表？

语法：列表对象名称 = [*元素1， 元素2， ……， 元素N*]



示例：创建列表对象的两种方法

方式 1：默认方法

列表对象 = [元素 1, 元素 2, 元素 3, …….. , 元素 N,]

```
>>> list1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,]
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list2 = [ ‘a’ , ’ b’ , ‘c’ , ‘d’ , ‘e’ , ‘f’ , ]
>>> list2
[ ‘a’ , ’ b’ , ‘c’ , ‘d’ , ‘e’ , ‘f’ ]
>>> list3 = [ ‘a’ , 1 , True , ‘Hello’ , ]
>>> list3
[ ‘a’ , 1 , True , ‘Hello’ ]
```

示例：创建列表对象的两种方法

方式 2：使用 range()内置函数

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。

列表对象 = list(range(stop))

```
>>> list1 = list(range(10))
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list2 = list(range(5, 10))
>>> list2
[5, 6, 7, 8, 9]
>>> list3 = list(range(0, 10, 2))
>>> list3
[0, 2, 4, 6, 8]
```

Python3 range()内置函数

- 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。

创建语法

① range(stop)

② 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）；
- stop: 计数到 stop 结束，但不包括 stop。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)

Python 列表

如何访问列表？

- 列表中值的切割也可以用到 **[头下标 : 尾下标 : 步长]** ，就可以截取相应的列表。
- 从左到右下标索引默认 0 开始，从右到左下标索引默认 -1 开始，下标可以为空表示取到头或尾。

代码演示： ch03-demo01-list-slice.py

```
# 定义列表
list1 = list(range(10))

# 输出列表
print(list1)
# 输出列表中的第一个元素
print(list1[0])
# 输出列表中的第二个元素和第三个元素
print(list1[1:3])
# 设置按步长为2截取列表
print(list1[0:10:2])
# 反向截取列表
print(list1[-5:-1])
```

运行结果：

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

能否对 mList 列表中的第一个元素进行修改？

列表更新

- 所谓的更新指的是对列表元素的 重新赋值、删除、添加等相关操作。

代码演示： ch03-demo02-list-update.py

```
# 定义列表
list1 = ['HTML', 'CSS', 'XML', 'DataBase',]

# 更新列表中的CSS更新为CSS3
list1[1] = 'CSS3'
print(list1)
# 删除列表中的DataBase
del list1[-1]
print(list1)
# 使用remove移除列表中的指定元素
list1.remove('XML')
print(list1)
# 向列表中添加一个元素
list1.append('Python')
print(list1)
# 向列表中添加一个子列表
list1.append(list(range(3)))
print(list1)
```

运行结果：

```
['HTML', 'CSS3', 'XML', 'DataBase']
['HTML', 'CSS3', 'XML']
['HTML', 'CSS3']
['HTML', 'CSS3', 'Python']
['HTML', 'CSS3', 'Python', [0, 1, 2]]
```

列表的操作符

- 在介绍字符串中已经了解过 + 和 * 号的作用，列表用法与其类似，同时也有 len( ) 和 in 等操作。

代码演示： ch03-demo02-list-operation.py

```
12  # 定义列表
13  list1 = list(range(5))
14  list2 = ['a', 'b', 'c']
15
16  # 获取列表中元素的个数
17  print('列表有{0}个元素'.format(len(list1)))
18  # 两个列表相加
19  print(list1 + list2)
20  # 两个列表相乘
21  print(list1 * 2)
22  # 判断列表中是否存在指定的元素
23  print(2 in list1)
```

运行结果：

```
列表有5个元素
[0, 1, 2, 3, 4, 'a', 'b', 'c']
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
True
```

嵌套列表

- 由于列表中可以放置各种类型的数据，因此也可以存放列表类型。

代码演示：

- >>> list_1 = [1, 2, 3, 4]
- >>> list_2 = range(5, 8)
- >>> list_3 = [list_1, list_2]
- [[1, 2, 3, 4], [5, 6, 7]]

列表中的函数

- 为了更好地操作列表对象，Python 也提供了很多的函数：

1) len(list) ：获取列表元素的个数；
2) max(list) ：获取列表中的最大值；
3) min(list) ：获取列表中的最小值；
4) list(seq) ：将元组对象转换成列表对象。

- 列表也提供了大量的方法：

1) .append(obj) ：在列表末尾添加一个元素；
2) .count(obj) ：统计某个元素在列表中出现的次数；
3) .index(obj) ：从列表中找出某个值第一个匹配的索引位置；
4) .insert(index, obj) ：向指定位置前序添加一个元素
5) .remove(obj) ：移除一个指定的元素；
6) .reverse() ：反向列表中的元素；

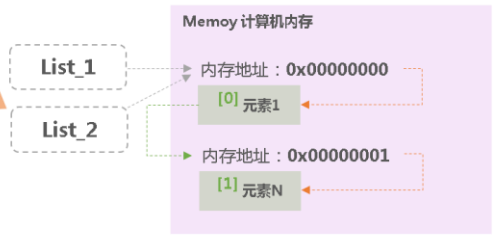
7） .sort() ： 对列表进行排序；

List[] 与 list[:] 的区别

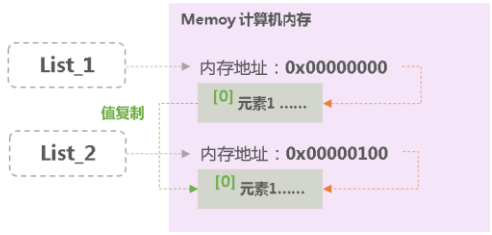
- “[ ]”标识，是典型的**引用操作**，**传址**操作
- “[:]”标识，是典型的**赋值操作**，**传值**操作

示例代码：

```
List_1 = [ 1, 2, 3, 4]
List_2 = list_1
```




```
List_1 = [ 1, 2, 3, 4]
List_2 = list_1[ : ]
```

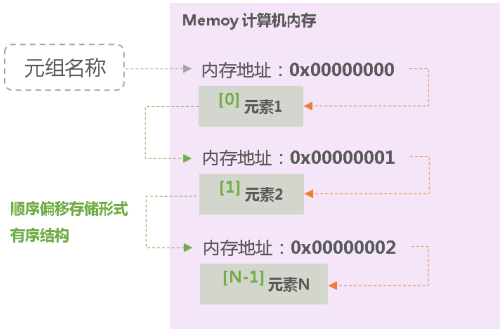


3.1.4 元组（Tuple）类型

- Tuple（元组）类似于 List（列表）。
- 元组不能二次赋值（元组内的元素*不允许更新*），相当于只读列表。
- 元组用“ ( ) ”标识。内部元素用逗号隔开。

如何创建元组？

语法：元组对象名称 = (*元素1*，*元素2*，.....，*元素N*)



元组的特点

- 元组与列表的所有操作基本类似，唯一不一样的地方是，元组的元素不允许被修改。

示例代码：

```
>>> tup1 = tuple(range(5))
>>> tup1
```

(0, 1, 2, 3, 4)

>>> tup2 = (5, 6, 7,)

>>> tup2

(5, 6, 7)

特别说明：

>>> tup1 = s(1)

>>> type(tup1)

<class ‘int’>

>>> tup1 = (1,)

>>> tup1

<class ‘tuple’>

3.1.5 字典（Dictionary）类型

- **字典**(dictionary)是除列表以外 python 之中最灵活的内置数据结构类型。
- 字典当中的元素是通过键来存取的，而不是通过偏移存取。
- 字典用“{ }”标识。字典由索引(key)和它对应的值 value 组成，是一个典型的“***k-v 值***”数据结构。

如何创建字典？

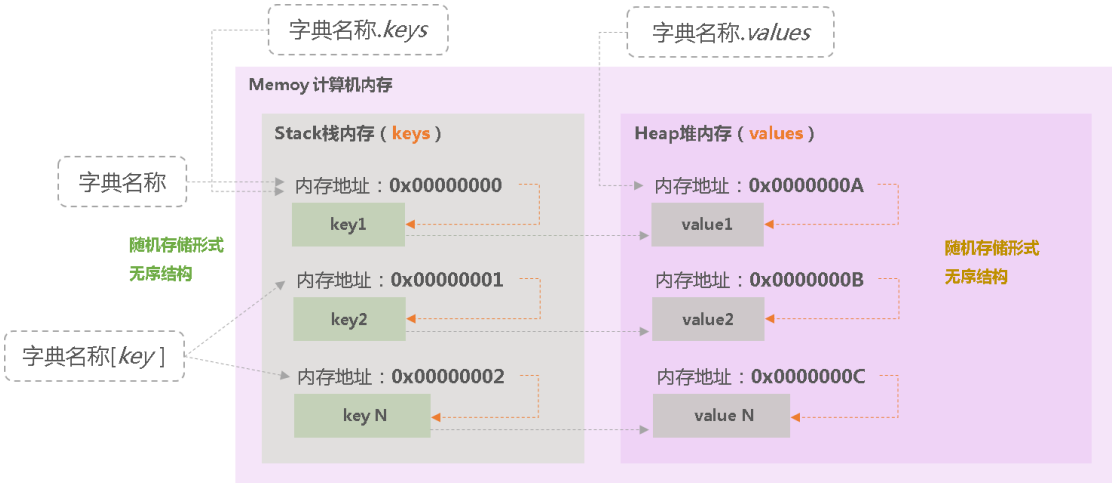
语法：

- 字典对象名称 = { }
- 字典对象名称 = { *key1* : *value1, key2* : *value2, …. , keyN* : *valueN* }

字典（Dictionary）k-v 值在内存中的表现形式

- **k-v 结构**一般情况下在操作访问的时候都会使用 key 索引进行每个元素的读取操作。
- 由于 key 索引键会被频繁访问，因此索引键 key 存放在 **Stack 栈**内存中，而 value 值则存储在 **Heap 堆**内存中。

内存表现形式：



如何访问字典？

语法：

- 字典对象名称[key]　　#访问 key 对应的 value 值
- 字典对象名称.keys　　#访问当前字典所有 key 索引键
- 字典对象名称.values　　#访问当前字典所有 valeus 值

代码演示： *ch03-demo05.py*

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技.CTO办公室"

#创建一个字典
mDict = {}
#创建一个字典元素
mDict['name'] = 'Teresa'
mDict[1] = 'She is a girl'
#创建一个字典
mDict2 = {'name':'teresa', 'age':18,
          'email':'teresa@chinasofti.com'}

#输出
print mDict['name']  #输出索引键'name'的值
print mDict[1]       #输出索引键'1'的值
print mDict2         #输出完整的字典
print mDict2.keys()    #输出所有的索引键
print mDict2.values()  #输出所有的值
```

运行结果：

```
Teresa
She is a girl
{'age': 18, 'name': 'teresa', 'email': 'teresa@chinasofti.com'}
['age', 'name', 'email']
[18, 'teresa', 'teresa@chinasofti.com']

***Repl Closed***
```

列表、元组和字典区别

列表（List）、元组（Tuple）和 字典（Dictionary）的异同点：

从承载数据类型的结构角度看：Python 的数据类型分为数字类型和非数字类型。

- 数字类型包括：整型、长整型、浮点型和复数型。
- 非数字类型包括：字符串、列表、元组和字典。

共同点：

1. 都可以使用切片、链接（+）、重复（*）、取值（a[ ]）等相关运算。
2. 截取方式相同：名称[头下标:尾下标]
3. 下标是从 0 开始算起，可以是正数或者负数，下标为空则表示取到头或者尾
4. 开始截取时，包含了下边界，而截取到最大范围不包括上边界。

不同点：

1. 列表 可以直接赋值
2. 元组不可以赋值
3. 字典按照 dict[k]=v 的方式赋值

3.1.6 集合（Set）类型

- 集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。
- 可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {} ；后者创建一个空的字典。

集合的创建

- obj1 = {1, 2, 3, 4, 5}
- obj2 = ({6, 7, 8, 9})

集合 Set 的创建

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # 删除重复的
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # 检测成员
True
>>> 'crabgrass' in basket
False
>>> # 以下演示了两个集合的操作
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # a 中唯一的字母
{'a', 'r', 'b', 'c', 'd'}
```

集合 Set 的关系操作（交、并、补）

```
s = set([3,5,9,10])       #创建一个数值集合
t = set("Hello")          #创建一个唯一字符的集合
```

```
a = t | s          # t 和 s 的并集
b = t & s          # t 和 s 的交集
c = t - s           # 求差集（项在 t 中，但不在 s 中）
d = t ^ s          # 对称差集（项在 t 或 s 中，但不会同时出现在二者中）
```

基本操作：

```
t.add('x')            # 添加一项
s.update([10,37,42])  # 在 s 中添加多项
```

3.1.7 数据类型转换

- 当需要对数据内置的类型进行转换，就需要了解 Python 数据类型的转换知识。
- 使用各种转换函数，转换函数返回一个新的对象，表示转换的值。
- **语法格式：**转换后变量 = 转换函数名称（待转换数据或变量）

主要类型转换函数参考表：

函数	描述
int(x [,base])	将x转换为一个整数
long(x [,base])	将x转换为一个长整数
float(x)	将x转换到一个浮点数
str(x)	将对象 x 转换为字符串
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
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
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

3.2 运算符

介绍 Python 中的七大与运算符，特别是 成员运算符和身份运算符。强调各种运算符的规则和标准。

3.2.1 Python 运算符

- 在我们日常生活中，经常用到 +、−、*、/ 这些统称为运算符。
- 操作数 与 运算符 的组合形成了表达式。

Python 中的运算符分为以下七大类：

- 算数运算符、比较（关系）运算符、赋值运算符、逻辑运算符
- 位运算符、成员运算符、身份运算符

3.2.2 算数运算符

- 假设 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

代码演示： ch03-demo05-operator01.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO办公室"

#声明变量
a = 21
b = 10
c = 0
```

Step2：进行算数运算

```
#四则运算
c = a + b
print "1 - c 的值为：", c
c = a - b
print "2 - c 的值为：", c
c = a * b
print "3 - c 的值为：", c
c = a / b
print "4 - c 的值为：", c
```

Step3：求模取余、幂运算和取整运算

```
#求模取余运算，获取余数
c = a % b
print "5 - c 的值为：", c
# 修改变量 a 、b 、c
a = 2
b = 3
c = a**b
print "6 - c 的值为：", c
a = 10
b = 5
c = a//b
print "7 - c 的值为：", c
```

运算结果：

```
1 - c 的值为：  31
2 - c 的值为：  11
3 - c 的值为：  210
4 - c 的值为：  2
5 - c 的值为：  1
6 - c 的值为：  8
7 - c 的值为：  2

***Repl Closed***
```

注意：Python3 里，整数除整数，只能得出整数。如果要得到小数部分，把其中一个数改成浮点数即可。

3.2.3 比较运算符

- 假设 a = 10 ， b = 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.
<>	不等于 - 比较两个对象是否不相等	(a <> b) 返回 true。这个运算符类似 != 。
>	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。

代码演示： ch03-demo06-operator02.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO办公室"

#声明变量
a = 21
b = 10
c = 0
```

Step2：进行比较运算

```
#进行比较运算
if ( a == b ):
   print("1 - a 等于 b")
else:
   print("1 - a 不等于 b")

if ( a != b ):
   print("2 - a 不等于 b")
else:
   print("2 - a 等于 b")

if ( a < b ):
   print("3 - a 小于 b" )
else:
   print("3 - a 大于等于 b")

if ( a > b ):
   print("4 - a 大于 b")
else:
   print("4 - a 小于等于 b")
```

运算结果：

```
1 - a 不等于 b
2 - a 不等于 b
3 - a 不等于 b
4 - a 大于等于 b
5 - a 大于 b

***Repl Closed***
```

3.2.4 赋值运算符

- 假设 a = 10 ， b = 20

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

代码演示： ch03-demo07-operator03.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技-CTO办公室"

#声明变量
a = 21
b = 10
c = 0
```

Step2：进行赋值运算

```
#进行赋值运算
c = a + b
print("1 - c 的值为：", c)
c += a
print("2 - c 的值为：", c)
c *= a
print("3 - c 的值为：", c)
c /= a
print("4 - c 的值为：", c)
c = 2
c %= a
print("5 - c 的值为：", c)
c **= a
print("6 - c 的值为：", c)
c //= a
print("7 - c 的值为：", c)

c1 = 12
c2 = 8
res = c1 ^ c2
print('res:' , res)
```

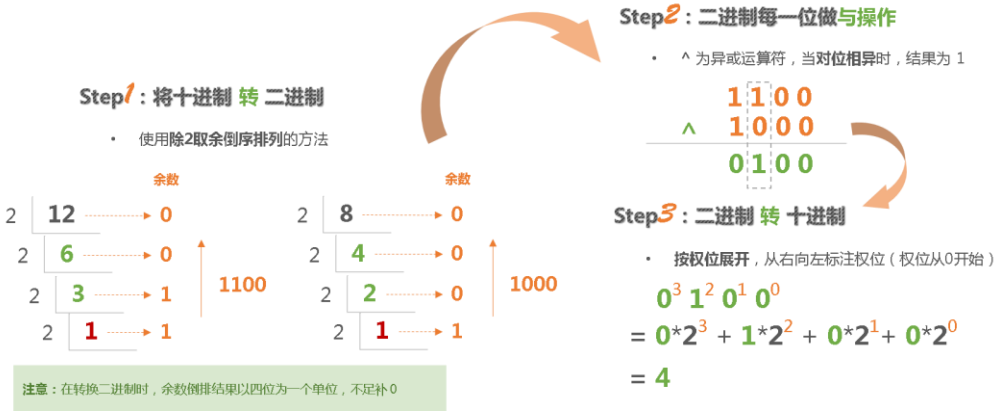
运算结果：

```
1 - c 的值为：  31
2 - c 的值为：  52
3 - c 的值为：  1092
4 - c 的值为：  52
5 - c 的值为：  2
6 - c 的值为：  2097152
7 - c 的值为：  99864

***Repl Closed***
```

3.2.5 位运算符

- 按位运算符是把十进制数字看作二进制来进行计算的。Python 中的按位运算法则举例如下：
- 设 a = 12 ， b = 8
- 求 res = a ^ b （*a 按位异或 b*）
- 最终 res = 4



- 假设 a = 60 ， b = 13

运算符	描述	实例
&	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。	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	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。	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1。	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	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。	(~a) 输出结果 -61 ，二进制解释： 1100 0011
<<	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。	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	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 输出结果 15 ，二进制解释： 0000 1111

代码演示： ch03-demo08-operator04.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技-CTO办公室"

#声明变量
a = 60              # 60 = 0011 1100
b = 13              # 13 = 0000 1101
c = 0
```

Step2：进行位运算

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

运算结果：

```
1 - c 的值为:  12
2 - c 的值为:  61
3 - c 的值为:  49
4 - c 的值为:  -61
5 - c 的值为:  240
6 - c 的值为:  15

***Repl Closed***
```

3.2.5 逻辑运算符

- 假设 a = 10 ， b = 20

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False ，x and y 返回 False ，否则它返回 y 的计算值。	(a and b) 返回 20。
or	x or y	布尔"或" - 如果 x 是非 0 ，它返回 x 的值，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True ，返回 False 。如果 x 为 False ，它返回 True。	not(a and b) 返回 False

3.2.6 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**，示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。

代码演示： ch03-demo10-operator06.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO办公室"

#声明变量
a = 10
b = 20
#声明列表
list = [1, 2, 3, 4, 5 ];
```

Step2：进行成员运算

```
#进行成员运算
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")
```

运算结果：

```
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

***Repl Closed***
```

3.2.7 身份运算符

- 身份运算符用于比较两个对象的存储单元。

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, 类似 id(x) == id(y)，如果引用的是同一个对象则返回 True，否则返回 False
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 ，类似 id(a) != id(b)。如果引用的不是同一个对象则返回结果 True，否则返回 False。

注意： id( ) 函数用于获取对象内存地址

代码演示： ch03-demo11-operator07.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO办公室"

#声明变量
a = 20
b = 20
```

Step2：进行身份运算

```
#进行身份运算
if ( a is b ):
    print("1 - a 和 b 有相同的标识")
else:
    print("1 - a 和 b 没有相同的标识")
# 修改变量 b 的值
b = 30
if ( a is not b ):
    print("2 - a 和 b 没有相同的标识")
else:
    print("2 - a 和 b 有相同的标识")
```

运算结果：

```
1 - a 和 b 有相同的标识
4 - a 和 b 没有相同的标识

***Repl Closed***
```

3.2.8 is 与 == 的区别

- 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。

代码演示： ch03-demo12-operator08.py

Step1：声明操作数变量

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO办公室"

#创建列表
list_1 = [1, 2, 3, 4]
```

Step2：进行身份运算

```
#创建列表
list_1 = [1, 2, 3, 4]

#将list_1的内存地址赋值给list_2
list_2 = list_1
#判断内存地址是否一致
print('list_1 和 list_2 内存地址一致: ',(list_1 is list_2))
print('list_1 和 list_2 值一致: ',(list_1 == list_2))
#将list_1的值传递给list_2
list_2 = list_1[:]
print('list_1 和 list_2 内存地址一致: ',(list_1 is list_2))
print('list_1 和 list_2 值一致: ',(list_1 == list_2))
```

运算结果：

```
list_1 和 list_2 内存地址一致:  True
list_1 和 list_2 值一致:  True
list_1 和 list_2 内存地址一致:  False
list_1 和 list_2 值一致:  True

***Repl Closed***
```

扩展 1/2：

Python 中会为每个出现的对象分配内存，哪怕他们的值完全相等（注意是相等不是相同）。如执行 a=2.0，b=2.0 这两个语句时会先后为 2.0 这个 Float 类型对象分配内存，然后将 a 与 b 分别指向这两个对象。所以 **a 与 b 指向的不是同一对象：**

示例代码：

```
a = 2.0
```

```
b = 2.0
a is b      # 结果为 False
a == b   # 结果为 True
```

扩展 2/2：

但是为了提高内存利用效率对于一些简单的对象，如一些数值较小的 int 对象，Python 采取重用对象内存的办法，如指向 a=2，b=2 时，由于 2 作为简单的 int 类型且数值小，Python 不会两次为其分配内存，而是只分配一次，然后将 a 与 b 同时指向已分配的对象：

示例代码：

```
a = 2
b = 2
a is b      # 结果为 True
a == b   # 结果为 True
```

但如果赋值的不是 2 而是教大的数值，情况就不样了：

示例代码：

```
a = 8888
b = 8888
a is b      # 结果为 False
a == b   # 结果为 True
```

3.2.9 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 or and	逻辑运算符

3.3 本章总结

通过本章节的学习，我们初步掌握了 Python 编程语言中的重要数据类型（序列），序列类型是 Python 编程语言核心数据类型，为我们后续的学习和对 Python 的使用打下了坚实的基础。运算符和表达式也是编程语言的重点，灵活地使用运算符可以大幅度提升我们的开发效率。

第04章：控制流语句（Ⅰ）

知识点

目标 1：了解控制流语句的作用

目标 2：掌握条件判断语句的使用

目标 3：掌握 while 循环语句的使用

目标 4：掌握 IDLE 断点调试技术

目标 5：掌握 Iterator 迭代器

技能点

实战任务 1：模拟用户登录验证

实战任务 2：菜单栏的生成

实战任务 3：用户信息录入显示

实战任务 4：城市信息输出

4.1 控制流语句

控制流语句即用来实现对程序流程的选择、循环、转向和返回等进行控制。

4.1.1 用途

- 控制语句可以用于控制程序的流程， 以实现程序的各种结构方式。

➢ 一般情况下，程序按照语句编写顺序依次执行，形成了一个标准的面向过程的结构化形式。但由于程序具备很强的逻辑性，有些时候我们需要根据某些条件选择性执行或跳跃执行某些语句。

➢ 当需要程序非顺序执行的时候，我们就需要控制流语句，改变其原有的顺序执行。

➢ 在 Python 中最常用的两种控制流语句是：**条件控制语句** 和 **循环控制语句**

4.1.2 条件控制语句介绍

条件控制语句可以根据是否满足自定义的条件选择性执行条件下的语句块。

应用场景举例：

- 用户登录，根据用于输入的账号或密码，使用条件判断是否与已有账户一致。

- 条件判断会使用到 if 关键字。
- 当然还可以嵌套多条件进行判断。

4.1.3 循环控制语句介绍

循环控制语句可以根据自定义或特定的有限次数进行同一语句块反复执行。

应用场景举例：

- 数据源读取数据，程序从数据源获取数据，重复封装数据的动作，循环的次数取决于数据源数据的个数。

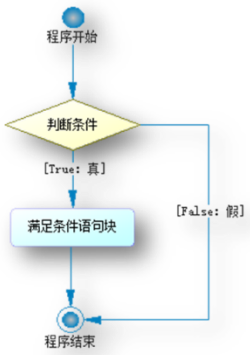
- 循环控制语句会使用到 while 或 for 关键字。
- 在 Python 中有两种比较典型的循环模式：while 循环 和 for 循环

4.2 条件控制语句

掌握条件控制语句的工作原理和基本实现语法，了解 if..elif…典型的结构以及多值判断的实现方法。

4.2.1 基本 if 语法介绍

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



基本语法：

```
if 条件表达式:
    条件语句块
    pass
```

pass 是Python中的关键字，代表一个空行（也可以不写），代表条件语句块的结束，也可以使用一个空行替代，提高代码结构可读性。

代码演示： ch04-demo01-if.py

```
12  # 声明一个变量并初始化
13  num = 10
14  # 条件判断
15  if num > 5:
16      print('num的值大于5')
17      pass
```

4.2.2 if…else… 结构

当不满足情况的时候我们可以使用 else 关键字实现逻辑结构。

场景模拟：用户输入姓名，如果输入的是“Monster”时，程序拒绝例会，否则程序输出“你好，

XXX”

基本语法：

```
if 条件表达式：
    条件语句块1
    pass
else:
    条件语句块2
    pass
```

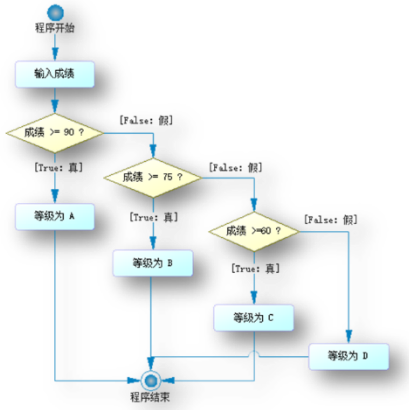
代码演示：ch04-demo02-ifelse.py

```
# 接受用户输入的姓名
name = input('请输入姓名:>')

# 判断用户名是否为monster
if name == 'monster':
    print('我不和怪物说话')
    pass
else:
    print('你好，{0}'.format(name))
    pass
```

4.2.3 多值判断 if…elif…else…

- 如果需要多个条件进行判断，Python 可以用 **elif** 关键字来实现 *（Python 不支持 switch 语句）*。
- **场景模拟：**成绩区间判断，若成绩[90 ， 100]为 **A**，成绩[75， 90）为 **B**，成绩[60 ， 75）为 **C**，成绩小于 60 为 **D**



基本语法：

```
if 条件表达式1：
    条件语句块1
    pass
elif 条件表达式2：
    条件语句块2
    pass
else:
    条件语句块3
    pass
```

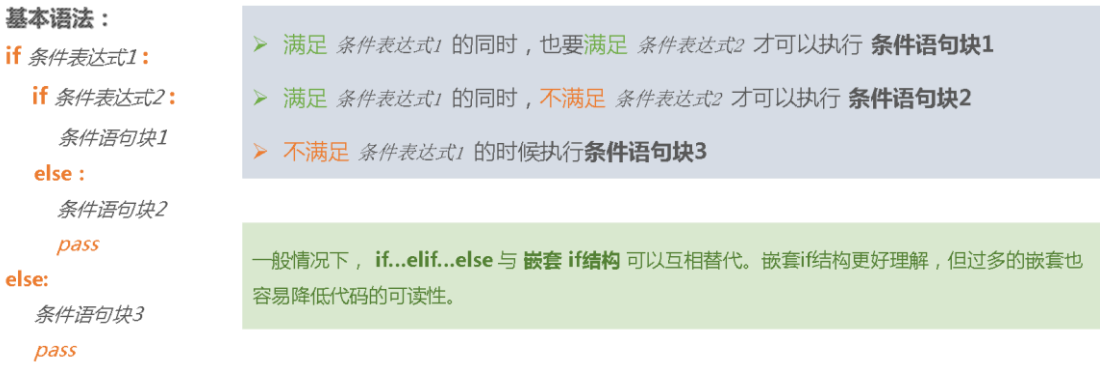
代码演示：ch04-demo03-ifelifelse.py

```
#等待用户输入成绩
score = int(input('输入您的成绩:>'))

#多值区间判断
if score >= 90:
    print('等级为 A')
    pass
elif score >= 75:
    print('等级为 B')
    pass
elif score >= 60:
    print('等级为 C')
    pass
else:
    print('等级为 D')
    pass
```

4.2.4 多值判断嵌套if结构

- 如果需要多个条件进行判断，Python 也可以用 **嵌套 if** 结构来实现。



4.2.6 实战任务：模拟用户登录

模拟系统通用功能登录模块

业务需求：

① 判断用户登录名称和密码是否为空，只要有一个为空则报错并结束程序。

② 判断用户登录名称和密码是否正确，两个必须同时正确，否则报错结束程序。

要求： 账号：admin 密码：123

技术需求：

① 控制台用户输入数据信息

② 使用条件控制语句进行数据提交前非空验证

③ 使用条件控制语句+条件运算符表达式进行数据验证

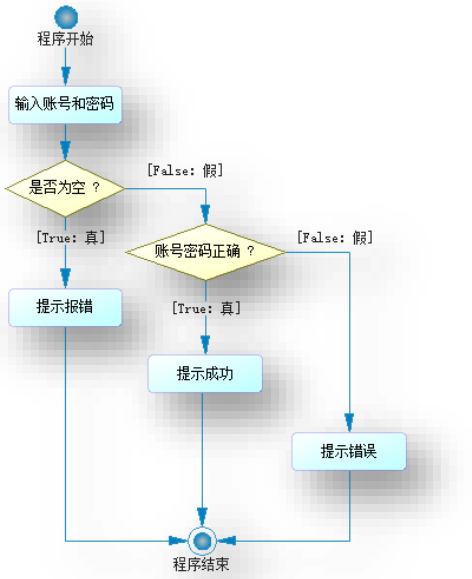
④ 保护用户输入的密码，在用户输入的时候不显示。

Step1：流程分析并绘制流程图，形成编程思路

步骤 1：使用 input()函数接收用户输入的登录账号和密码

步骤 2：使用 if 条件判断账号和密码是否为空，由于业务要求只要有一个为空就报错，则确定逻辑运算符为 OR

步骤 3：使用 elif 判断账号和密码是否正确，由于业务要求账号和密码必须同时正确才可登录，则确定逻辑运算符为 AND



Step2：编码实现业务逻辑 ch04-demo04-login.py

步骤 1： 使用 input()函数接收用户输入的登录账号和密码

```
# 控制台接受用户的输入
account = input('登陆账号:>')
password = input('登陆密码:>')
```

步骤 2、3： 非空验证 并 进行正确性验证

```
# 进行非空验证
if account == '' or password == '':
    print('错误：账号密码不能为空!')
    pass
else:
    # 判断账号密码是否正确
    if account == 'admin' and password == '123':
        print('欢迎登陆，{0}'.format(account))
        pass
    else:
        print('错误：账号密码错误，请核实后再登陆。')
        pass
```

步骤 4： 设置密码保护（输入不显示）

导入 getpass 模块，使用 getpass()函数接受控制台输入

```
11  # 导入模块
12  import getpass
13  # 使用getpass()方式接受控制台输入
14  password = getpass.getpass('登陆密码:>')
```

思 考

下列代码执行结果是什么？

示例代码 1：

```
a = 0
b = 1
if (a>0) and (b/a>2):
    print ‘yes’
else:
    print ‘no’
```

运行结果：no

示例代码 2：

```
a = 0
b = 1
if (a>0) or (b/a>2):
    print ‘yes’
else:
    print ‘no’
```

运行报错！

分析：Python *复合布尔表达式* 计算采用 **短路规则**，即如果通过前面的部分已经计算出整个表达式的值，则后面的部分不再计算。

4.3 循环控制语句

4.3.1 循环控制语句

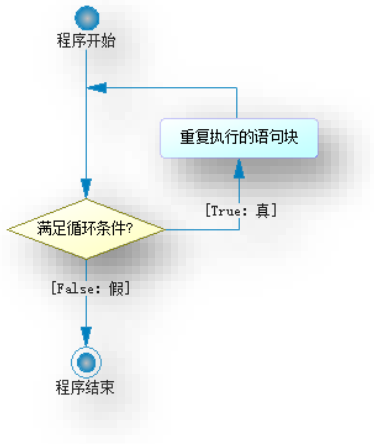
- 程序在一般情况下是按顺序执行的，编程语言提供了各种控制结构，允许更复杂的执行路径。
- 循环语句允许执行一个语句或语句组多次，下面是在大多数编程语言中的循环语句的一般形式：

循环基本规则：

① 需要循环变量（可以为数字类型，字符或字符串类型等）；

② 需要有循环条件（可以为表达式或布尔值等）；

③ 循环语句块中修改循环变量（变量趋近于不满足循环条件方向，否则为无限循环）。



4.3.2 循环控制语句的分类

循环类型	描述
while 循环	在给定的判断条件为 true 时执行循环体，否则退出循环体。
for 循环	重复执行语句
嵌套循环	你可以在while循环体中嵌套for循环

注意：Python 语言中没有 do…while 循环

4.3.3 循环控制语句中的关键字

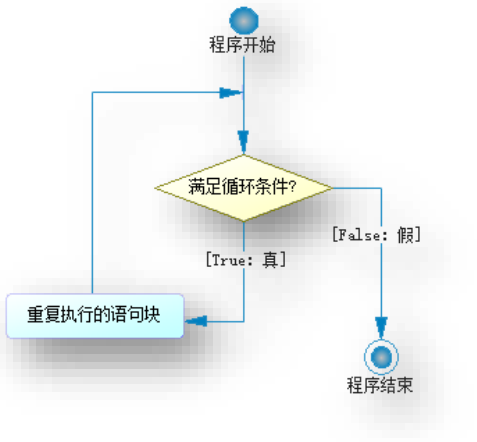
- 循环控制语句可以更改语句执行的顺序。Python 支持以下循环控制语句关键字：

循环类型	描述
break	在语句块执行过程中终止循环，并且跳出整个循环
continue	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。

4.3.4 While 循环控制语句

- Python 编程中 while 语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。其基本形式为：

```
while 判断条件:
    执行语句块
pass
```



示例：使用 while 循环输出数字 0~9

首先，问自己三个问题：

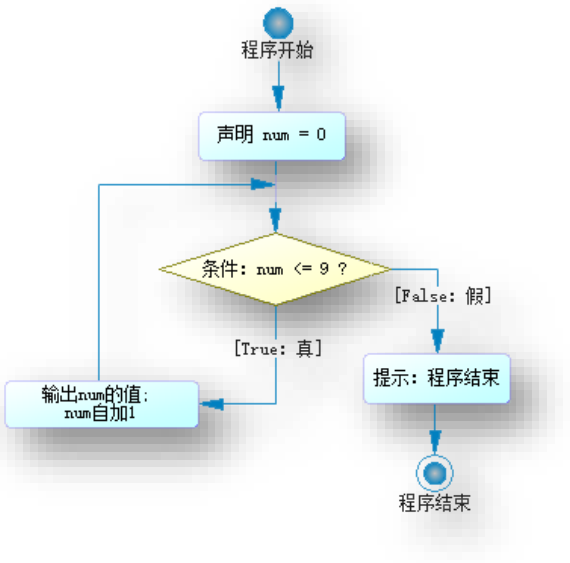
- **问题 1**：需要重复执行什么？
- **问题 2**：重复执行的条件是什么？
- **问题 3**：如何结束重复操作？

流程实现：

- **首先**，创建循环变量 num，同时该变量也是程序的操作变量。

- **其次**，确定循环条件，根据要求只要 num 小于等于 9 就可以输出。
- **再次**，确定循环体需要重复执行输出 num 的语句
- **最后**，在循环体中一定要体现修改循环变量的值 num 自加 1（趋于不满足条件方向）

Step1：流程分析并绘制流程图



Step2：编写代码实现 ch04-demo05-while.py

```
12  # 声明循环变量
13  num = 0
14
15  # 使用while循环
16  while num<=9:
17      # 输出num的值
18      print('num:> %d' %num)
19      # 循环变量自增1
20      num += 1
21      pass
22
23  # 提示
24  print('程序结束')
```

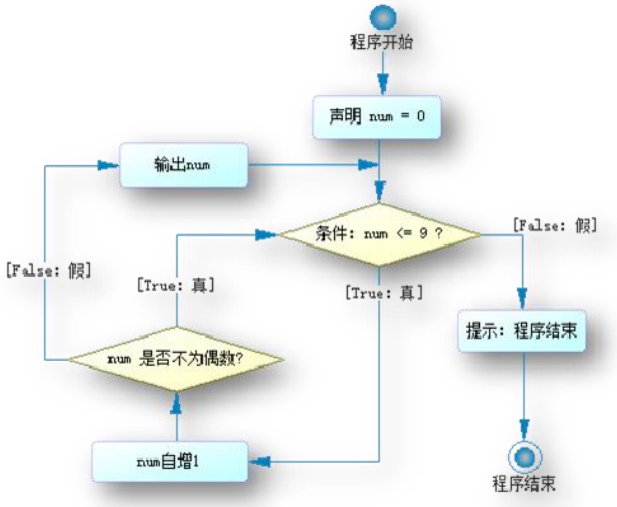
```
num:> 0
num:> 1
num:> 2
num:> 3
num:> 4
num:> 5
num:> 6
num:> 7
num:> 8
num:> 9
程序结束
```

4.3.5 break 和 continue

- while 循环语句时还有另外两个重要的关键字 *continue* 和 *break* 来跳过循环，该关键字出现在循环体当中，用来改变程序流。
- **break** 用于终止并退出循环，执行循环语句块之后的代码。
- **continue** 用于终止当前正在执行的循环，程序流回到循环条件判断，进行下一次循环。

示例：输出数字 1~10 中的偶数

Step1：流程分析并绘制流程图



Step2：编写代码实现 ch04-demo06-continue.py

```python
# 声明一个变量
num = 1

# 使用while循环
while True:
    # 判断是否到10
    if num > 10:
        # 跳出循环并终止
        break
    # 判断当前数字是否为偶数
    if num % 2 != 0:
        num += 1 # 自增1
        # 跳出当前循环，进入到下一次循环中
        continue
        pass
    else:
        print('num:> %d' %num) # 输出
        num += 1 # 自增1
        pass
    pass
```

4.3.6 无限循环

当 while 循环条件为 永真值 时 或 循环变量趋向于不满足循环条件发展时，则出现永无休止的执行循环体，我们称之为 无限循环。

无限循环可能出现的情况：

```
while 1==1 :
    执行语句块
    pass
```

```
while True :
    执行语句块
    pass
```

```
a = 0
while a <5 :
    执行语句块
    a = a - 1
    pass
```

- 在某些场景中，我们恰恰需要无限循环这种情况（比如 菜单操作等）。
- **注意：**当程序出现无限循环情况，若要终止程序 u 运行可以使用快捷键 Ctrl+C

4.3.7 实战任务：菜单栏的生成

使用 while 循环生成菜单栏

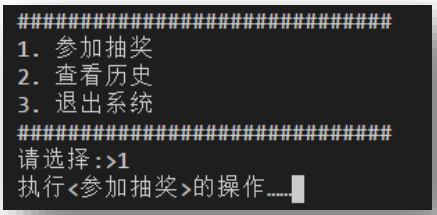
业务需求：

① 显示菜单栏中的选项；

② 用户输入选项编号；

③ 若出现编号不存在则提示用户重新输入。

运行结果：



技术需求：

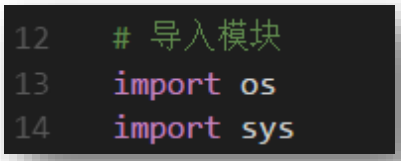
① While 循环的使用；

② Contimue 关键字的使用；

③ 系统退出指令。

编码实现 ch04-demo07-menu.py

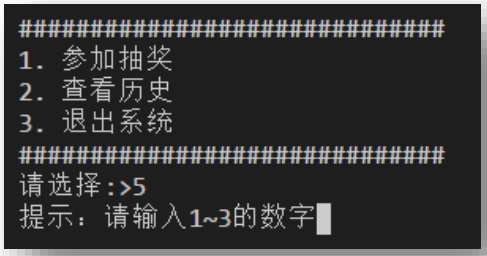
步骤 1：导入模块



步骤 2：显示菜单并进行输入验证

```
16  # 使用while循环实现菜单栏
17  while True:
18      os.system('cls') # 清屏操作
19      print('#' * 30)
20      print('1. 参加抽奖')
21      print('2. 查看历史')
22      print('3. 退出系统')
23      print('#' * 30)
24      # 接受用户输入选择
25      choice = int(input('请选择:>'))
26      # 判断用户输入是否为1~4之间的数字
27      if choice not in range(1,4):
28          input('提示：请输入1~3的数字')
29          os.system('cls') # 清屏操作
30          continue # 跳出当前循环
31          pass
32      pass
```

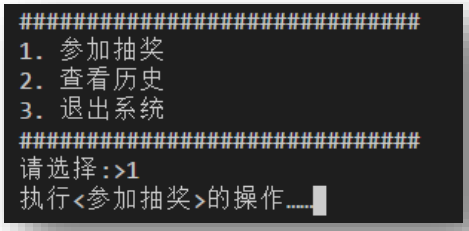
运行结果：



步骤 3：添加其他选项的判断操作

```
# 判断用户选择的序号
if choice == 1:
    input('执行<参加抽奖>的操作……')
    continue # 跳出当前循环
    pass
elif choice == 2:
    input('执行<查看历史>的操作……')
    continue # 跳出当前循环
    pass
else:
    choice = input('确定退出系统吗(y/n)?')
    if choice.lower() == 'y':
        sys.exit(0) # 正常退出系统
        pass
    else:
        continue # 跳出当前循环
        pass
```

运行结果：



4.3.8 循环使用while…else语句

- 在 python 中，while … else 在循环条件为 false 时执行 else 语句块：

```
# 声明变量
count = 0
# while循环
while count < 5:
    # 输出
    print('%d 小于 5' %count)
    count += 1 # 自增1
    pass
else:
    print('%d 大于 5' %count)
    pass
```

运行结果：

```
0 小于 5
1 小于 5
2 小于 5
3 小于 5
4 小于 5
5 大于 5
```

4.3.9 实战任务：用户信息录入

使用 while 循环+条件判断实现用户姓名的录入和输出

业务需求：

① 用户录入姓名信息。

② 每次录入完毕后需要提示是否继续？

③ 当用户终止录入时，格式化输出用户姓名并显示录入人数。

运行结果：

```
开始录入用户姓名
用户姓名:>张三丰
是否继续?(y/n):>y
用户姓名:>张翠山
是否继续?(y/n):>y
用户姓名:>张无忌
是否继续?(y/n):>n
录入终止
------------------------------
总共录入{3}条记录
第{1}个用户姓名：张三丰
第{2}个用户姓名：张翠山
第{3}个用户姓名：张无忌
```

技术需求：

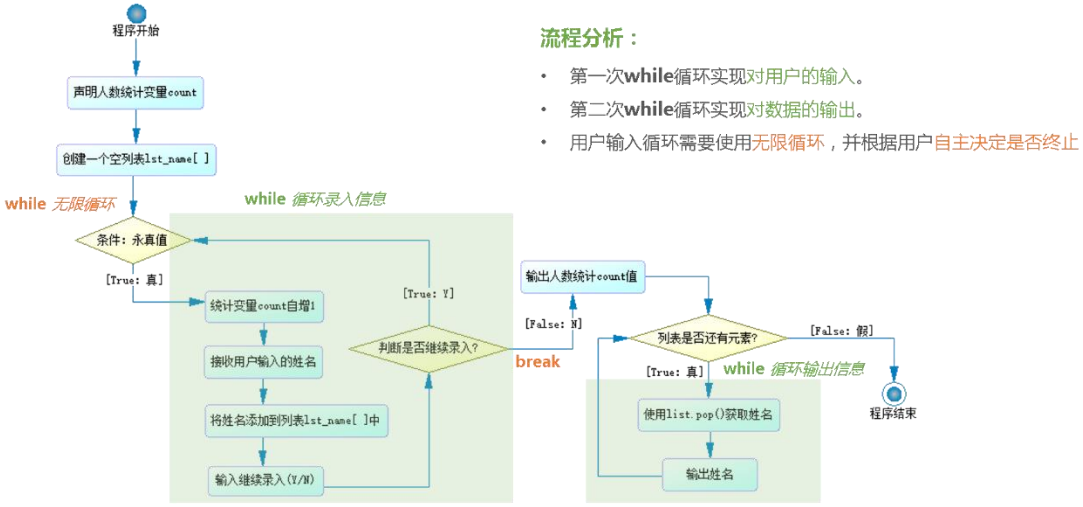
① 使用列表临时存储用户录入的姓名数据

② 使用 while 循环实现重复录入并提示的相关操作

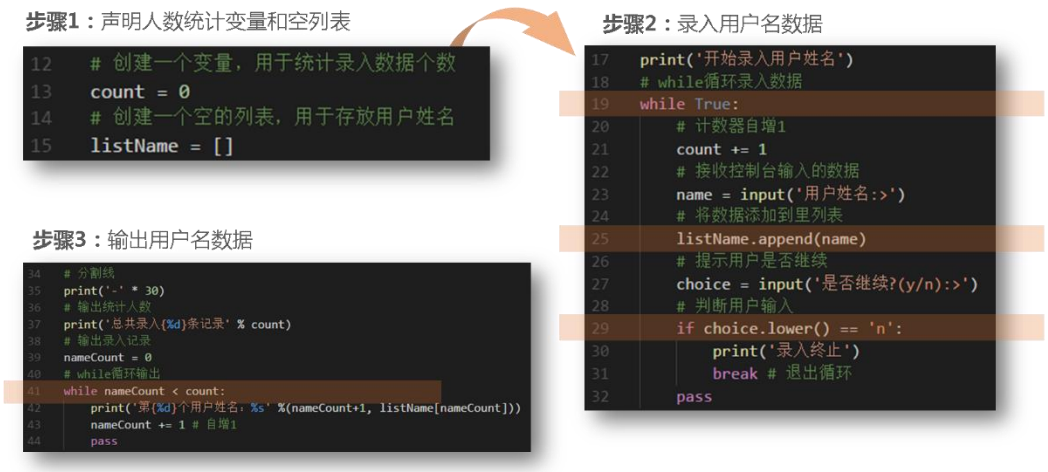
③ 使用 if 判断用户是否终止录入？

④ 使用 while 循环输出已录入姓名信息

Step1：流程分析并绘制流程图，形成编程思路



Step2：编码实现 ch04-demo09-users.py



4.4 iterator 迭代器

4.4.1 迭代器的语法

- 迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。
- 迭代器是一个可以记住遍历的位置的对象。
- 迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。
- 迭代器有两个基本的方法：**iter()** 和 **next()**。
- 字符串，列表、元组和字典对象都可用于创建相应的迭代器。

代码演示：

```
>>> list1 = list(range(2))  # 创建一个列表
>>> listIter = iter(list1)  # 创建一个列表迭代器
>>> next(listIter)  # 输出第 1 个元素
0
>>> next(listIter)
1
```

说明：若 next()调用超出显示范围则抛出异常 StopIteration

4.4.2 实战任务：城市信息显示

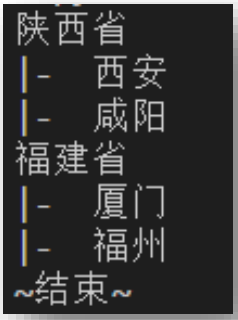
使用 while 循环 + Iterator 迭代器 实现对城市信息的输出

业务需求：

① 读取省份城市数据集；

② 按层次输出省份->城市名称。

运行结果：



技术需求：

① 使用字典临时存储省份和城市的信息；

② 使用 while 循环实现输出相关信息；

③ 使用迭代器逐条信息输出。

④ 使用异常处理

实现步骤：

步骤1：创建省份城市字典数据

```
# 创建一个城市信息数据集
dictCities = {'陕西省':['西安','咸阳'],
              '福建省':['厦门','福州']}
```

步骤2：使用while循环输出

```
# 使用Iterator输出信息
provincesIter = iter(dictCities)
try:
    while True:
        # 输出省份名称
        province = next(provincesIter)
        print(province)
        pass
except StopIteration:
    print('~结束~')
```

步骤3：输出城市信息

```
# 输出省份中的城市
citiesIter = iter(dictCities[province])
try:
    while True:
        city = next(citiesIter)
        print('|- ', city)
        pass
except StopIteration:
    pass
```

4.5 断点调试

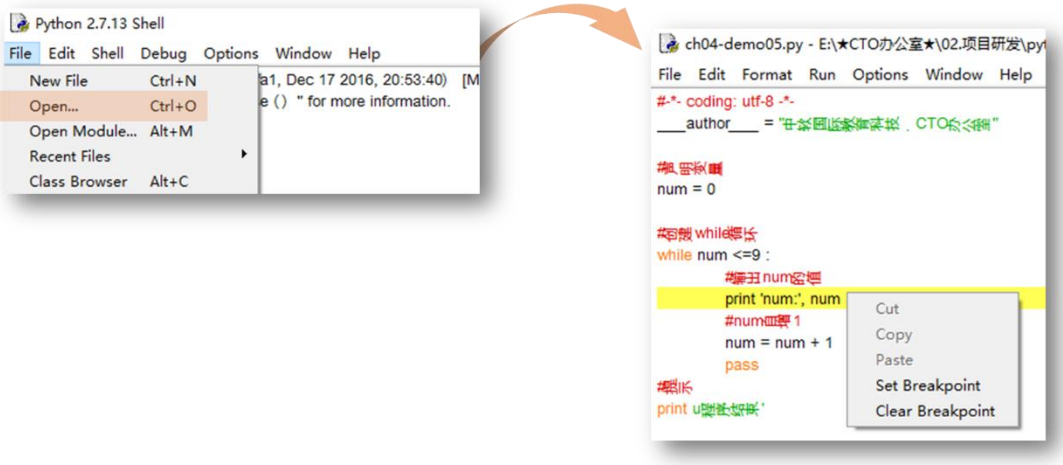
4.5.1 断点调试的意义

- 在程序开发过程中，我们经常要进行开发测试，以保障我们提交的程序正确性。
- 断点调试技术是在编写程序中很重要的一种开发测试技术。

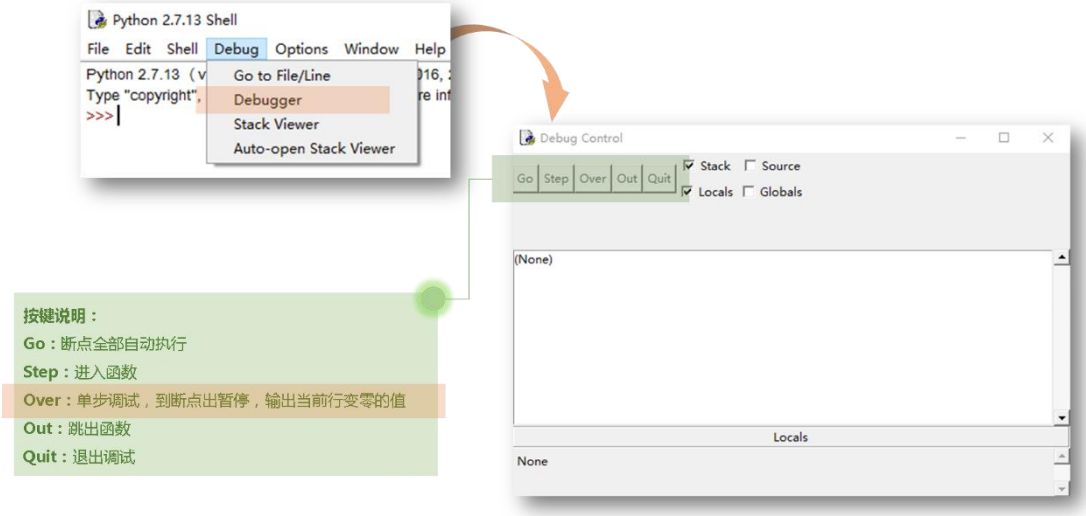
4.5.2 IDLE 进行断点调试

Step1：打开 IDLE 工具（Python Shell），选择需要调试的文件

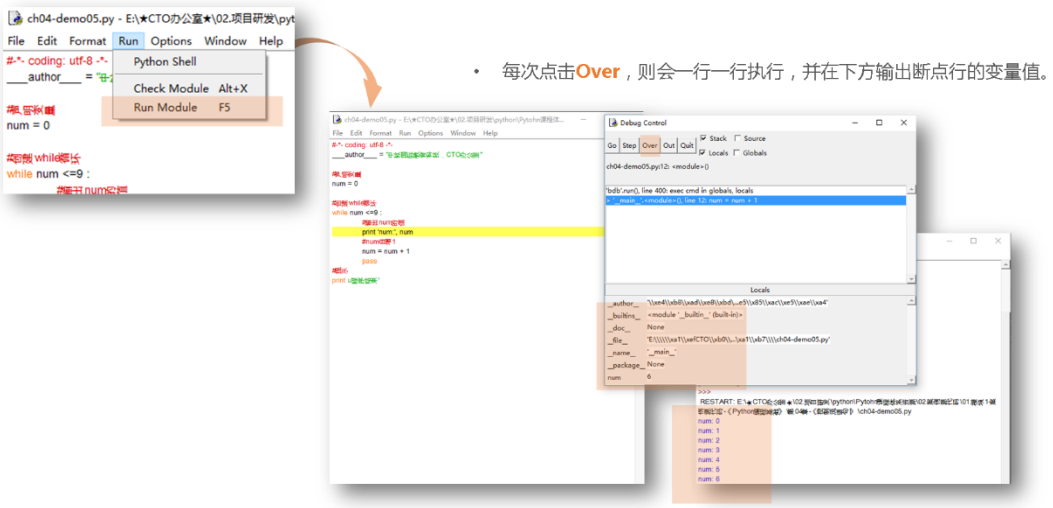
Step2：右击需要设置断点的代码行，选择 Set Breakpoint



Step3：在 IDLE 工具（Python Shell）中选择 Debugger 调试器



Step4：在打开的 py 文件窗口中，选择 Run Module 运行



4.6 本章总结

通过本小节的学习，我们掌握和了解了 Python 编程语言中的控制流语句，控制流语句是软件编程过程中非常重要的语法结构，本章通过对 while 循环的详细介绍，让大家初步认知和了解循环在编程中的应用。

第05章：控制流语句（II）

知识点

目标 1：了解 for 循环的作用和基本语法

目标 2：掌握 for 循环的基本语法和使用技巧

目标 3：掌握嵌套 for 循环的使用

目标 4：推导式的使用

目标 5：错误及异常处理

技能点

实战任务：银行金额汉字转换

5.1 再看 range 数据类型

了解 range 的类型的一些基本操作

5.1.1 range类型

- **range** 也是一种类型（type），它是一个数字的序列（s sequence of numbers），而且是不可变的，通常用在 for 循环中
- 每次使用 range( )默认返回一个列表对象。

实现原理：

>>> mlist = range(10) *说明:* 首先在内存中构建一个列表对象再并将 0~9 数字添加到列表中。

创建语法：

- **range**(*stop*)：**list** #创建一个列表，默认从 0 开始到 *stop* 指定范围的前一个结束，每次递增步长为 1。
- **range**(*[start,] stop [,step]*)：**list** #创建一个列表，从 *start* 开始到 *stop* 指定范围的前一个结束，每次按照 *step* 指定的步长递增。

5.1.2 range（）示例

使用 **range（）**输出 0~9 数字 。

- >>> range(10)
- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

- >>> range(0, 10)
- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

当 step 为正时，一个 range 的元素值为 r[i] = start + i * step 且 r[i] < stop； step 为负时，r[i] > stop。

使用 **range（）**输出 1~9 之间的奇数 。

>>> range(1, 10, 2)

[1, 3, 5, 7, 9]

使用 **range（）**输出 0~-10 之间的偶数 。

>>> range(0, -10, -2)

[0, -2, -4, -6, -8]

5.2 For 循环语句

掌握 for 循环的基本语法格式，了解 for 循环的应用场景

5.2.1 for循环语句

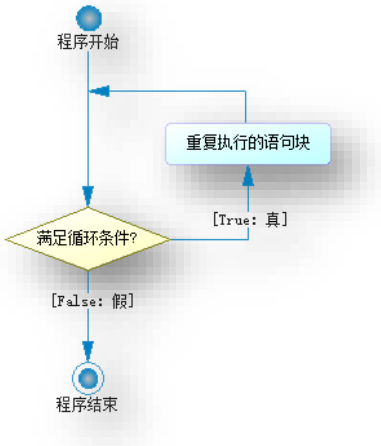
- **For 循环语句**一般用于已知循环次数的遍历操作，比如列表、字符串等。

基本语法：

for *变量* **in** *循环对象* **：**

循环语句块;

pass



5.2.2 for循环使用说明

- 使用 for 循环输出 0~9 数字：ch05-demo01-for-basic.py

```
12    # for循环输出0~9的数字
13    for num in range(10):
14        print('num: %d' % num)
15        pass
```

- 使用 for 循环输出 9~0 数字：ch05-demo02-for-basic.py

```
12    # for循环输出0~9的数字
13    for num in range(9, -1, -1):
14        print('num: %d' % num)
15        pass
```

- 输出列表元素：ch05-demo03-for-list.py

```
12    # 创建一个列表数据
13    mList = list('Python')
14
15    # for循环输出0~9的数字
16    for i in mList:
17        print(i)
18        pass
```

- 反向输出列表元素：ch05-demo04-for-list.py

```python
# 创建一个列表数据
mList = list('Python')

# for循环输出0~9的数字
for i in mList[::-1]:
    print(i)
    pass
```

For 可以遍历所有的序列对象 ch05-demo05-sequences.py 到此，我们基本掌握了 For 循环的操作！

5.2.3 嵌套for循环

- 嵌套循环常指两个 for 循环叠加使用。外层循环循环 1 次，内层循环循环 N 次。
- 例如：分针每走 1 格（外层循环），秒针需要走 60 格（内层循环）

示例代码：输出九九乘法表

代码演示： ch05-demo06-jiu.py

```python
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO办公室"

#外层循环控制乘数
for i in range(1,10):
    #内层循环控制被乘数
    for j in range(1,i+1):
        res = i * j
        print '%d*%d=%d' %(j,i,res),
    print ''
```

运行结果：

```
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

***Repl Closed***
```

5.2.4 for…else…语句

- 在 Python 中，**for … else** 表示这样的意思，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，与上一章中的 while … else 是一样。

场景模拟：输出 1~10 之间的质数并打印整除过程。*（质数又称为素数，特指只能被 1 和自己整除的数字）*

```python
# 创建一个列表对象
mList = ['apple', 'pear', 'banana']

# 请输入查找的水果
f = input('请输入水果的名称:>')

# 循环遍历水果列表
for fruit in mList:
    if fruit == f:
        print('找到了{0}'.format(fruit))
        break
else:
    print('没有找到您要的水果')
```

5.2.5 实战任务：银行金额大写汉字转换

使用 For 循环实现数字转换成大写汉字算法

业务需求：

- 银行电子支票业务在金额部分需要使用大写的汉字，因此需要将用户录入的数字信息转变为汉字。
- 目前只需完成 1~5 位整数转换即可。

示例：

输入金额:> 32542

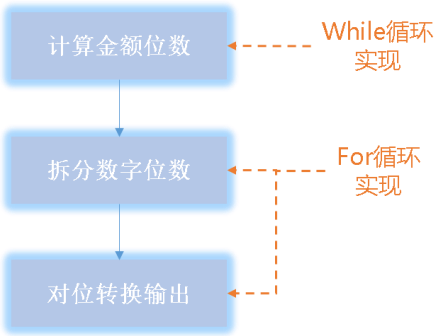
汉字转换:> 叁 萬 贰 仟 伍 佰 肆 拾 贰 圆 整

技术需求：

- 使用 **For 循环**完成数字每一位的**拆解**。
- 利用**列表下标**实现对位转换。

Step1：编程思路

- 程序可以拆分为 **3 个环节**实现：
    - 环节 1：计算出用户输入金额的位数；
    - 环节 2：利用已知位数完成每一位的拆解；
    - 环节 3：通过列表下标对位实现最终输出。



开发技巧：

- 需要创建两个列表，为后续对位转换做准备：
- 汉字列表：['零', '壹', '贰', '叁', '肆', '伍', '陆', '柒', '捌', '玖', '拾']
- 单位列表：['圆','拾', '佰', '仟', '萬']

Step2：编码实现 ch05-demo09-bank.py

- **步骤 1**：声明两个列表（*汉字列表* 和 *单位列表*）

```
12  # 前期准备汉字数字的列表
13  a = ["零", "壹", "贰", "叁", "肆", "伍", "陆", "柒", "捌", "玖", "拾"]
14  # 前期准备汉字的单位列表
15  b = ["圆","拾", "佰", "仟", "萬"]
```

Code 说明：

- [行：13] 列表 a 为汉字列表，我们会发现每个汉字恰恰是该列表的元素下标。这样设计的好处在于如果我们拆分出来的数字是 5，那么对位转换就可以使用 a[5]。那么，返回的值为 “伍”。
- [行：15] 列表 b 为单位列表，我们的第一个元素 b[0] = ‘圆’，b[1] = ‘拾’ 依次类推……这样设计的目的是为了让我们在输出的时候可以对位单位。

步骤 2：使用等待输入接收用户输入的金额数字

```
16  # 步骤1：编写代码接受控制台输入的5位以内的数字
17  # Tip: Python中使用 var = input("文字提示符")
18  # Tip: 数据类型转换
19  money = int(input("请输入5位数字金额:>"))
20  #print(type(money))
21  # 创建money的副本变量m1
22  m1 = money
```

Code 说明：

- ［行：19］使用 input()函数，它会将用户输入的字符串 “1234”转换成整型 1234 并赋值给 money。
- ［行：22］m1 的作用是制作一个 money 的副本，当 money 在进行拆解重组的时候可以保留原有用户数据。

步骤 3：计算用户输入的金额位数

实现整型数字位数计算的算法很多，我们在这里使用整除 10 的方法，并用 while 循环实现。

```
24  # 步骤2：计算出输入数字的位数（几位？）
25  # 创建一个循环计数器（用于统计输入数字的位数）
26  count = 0
27  # 使用while循环进行数字的循环拆分（数字整除取商值）
28  while money>0:
29      # 使用money整除10并使用int类型转换取得商值
30      money = int(money/10)
31      # 计数器+1
32      count += 1    # count = count + 1
33      pass
```

Code 说明：

- ［行：26］count 变量用于存储最终计算出来的位数，它其实是一个累加器。
- ［行：30］我们使用数字整除 10 的办法计算出数字一共几位组成，数字每次整除 10 后将结果重新覆盖 money 变量。
- ［行：32］当最终商 0（即结果为 0）时停止循环。而 count 数字累加器则统计出一共有几位。

步骤 4：拆分金额的每一位数字，并存放到一个列表中保存

个位：(213/1)%10
十位：(213/10)%10
百位：(213/100)%10
.....
公式： (数字 / $10^{位数幂}$) %10

代码实现：

```
37  # 步骤3：拆分出每一位上面的数字
38  # 创建一个计算因数
39  c = 10
40  # 创建一个空列表，用于存放我们的拆位的数字
41  mlist = []
42  # 使用for循环遍历数字并实现按位拆分
43  for i in range(0, count):
44      # 拆分位数
45      res = m1/(c**i)%10
46      # 将拆分的每一个位数放入到空列表中
47      mlist.append(int(res))
48      pass
```

Code 说明：

- [行：43] for 循环的次数为 *步骤 3* 中计算出来的金额位数 count
- [行：45] （数字 / $10^{位数幂}$）%10 转换成 Python 语法为：（数字/10**位数）%10
- [行：47] 将拆分出来的位数存放到 mlist 列表中，使用 append( )压栈的结果是个位在先，*即 mlist* [3, 1, 2]

步骤 5：对位转换格式化输出

比如，拆分数字 **213** 每一位

实际金额：**￥213**
列表存放：**mlist [3, 1, 2]**
For循环反向输出 ←
- 利用mlist元素做为a[]的下标，对位获取汉字；
- 使用b[金额位数-1]对位获取单位

代码实现：

```
52  # 步骤4：进行对位输出
53  print("转换汉字为:> ", end='')
54  # 使用for循环反向输出拆位数结果
55  for j in mlist[::-1]:
56      # 对位输出
57      print(a[j], b[count-1], end='')
58      count -= 1
59      pass
60  print("整")
```

Code 说明：

- [行：55] 使用 mlist[::-1]实现反向输出列表元素，保证与用户输入一致。
- [行：57] 使用汉字对位 连接 单位对位 输出。end=‘’，这种语法格式代表输出不换行。

5.3 推导式

Python3 中重要的应用方式，介绍列表、字典以及集合推导式的应用

5.3.1 推导式介绍

推导式 comprehensions（又称解析式），是 Python 的一种独有特性。推导式是可以从一个数据序列构建另一个新的数据序列的结构体。 共有三种推导，在 Python2 和 3 中都有支持：

① 列表(list)推导式

② 字典(dict)推导式

③ 集合(set)推导式

5.3.2 列表推导式

基本格式

```
variable = [out_exp_res for out_exp in input_list if out_exp == 2]
```

- 语法说明
  - out_exp_res: 列表生成元素表达式，可以是有返回值的函数。
  - for out_exp in input_list: 迭代 input_list 将 out_exp 传入 out_exp_res 表达式中。
  - if out_exp == 2: 根据条件过滤哪些值可以。

示例应用 1

代码演示 1： ch05-demo10-for-comprehensions.py

```
>>> mlist1 = [i for i in range(1,101) if i % 2 == 0]
>>> mlist1
```

代码演示 2： ch05-demo10-for-comprehensions.py 数字拆位

```
>>> number = 4321
>>> mlist2 = [int(number/(10**i)%10) for i in range(0, 4)]
>>> mlist2.reverse()
>>> mlist2
```

示例应用 2

代码演示 3： 使用（）替代原先的[ ] 将会得到一个 generator 生成器对象

```
# 得到生成器对象
list1 = 'python'
generator = (ord(i) for i in list1)
print(type(generator))
# 输出生成器对象
```

```
genIter = iter(generator)
for i in range(len(list1)):
    print(next(genIter))
```

5.3.3 字典推导式

- 字典推导和列表推导的使用方法是类似的，只不中括号该改成大括号。

代码演示 4：ch05-demo10-for-comprehensions.py 将 key 值字母相同的值进行合并累加

```
# 字典推导式
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}
mcase_frequency = {
          k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
          for k in mcase.keys()
          if k.lower() in ['a','b']
}
print(mcase_frequency)
```

字典推导式

- K-V 值互换

代码演示 5：ch05-demo10-for-comprehensions.py

```
# 字典推导式
mcase = {'a': 10, 'b': 34}
mcase_frequency = {v: k for k, v in mcase.items()}
print(mcase_frequency)
```

5.3.4 集合推导式

- 跟列表推导式也是类似的。 唯一的区别在于它使用大括号{}。

代码演示 5：ch05-demo10-for-comprehensions.py

```
# 集合推导式
>>> square= { i**2 for i in range(10) if i % 2 == 0]}
>>> square
```

5.4 异常处理

Python3 中异常的概念以及如何在程序中使用异常处理

5.4.1 Python3的错误和异常

- 在刚学习 Python 编程时，经常会看到一些报错信息，在前面我们没有提及。
- Python 有两种错误很容易辨认：语法错误 和 异常。

程序语法错误：

Python 的语法错误或者称之为解析错误，在开发之初经常遇到。

```
>>> num = range(5)
>>> num[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
>>> num[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: range object index out of range
```

语法错误 Error

- Python 的语法错误或者称之为解析错误，在开发之初经常遇到。

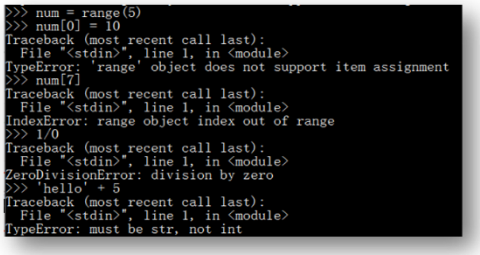
```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

- 这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号（:）。
- 语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

5.4.2 异常 Except

即便 Python 程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里：



- 异常以不同的类型出现，这些类型都作为信息的一部分打印出来： 例子中的类型有 ZeroDivisionError，NameError 和 TypeError。
- 错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

5.4.3 异常的处理的关键字及语法结构

在 Python3 中使用 **try…except…finally** 关键字处理异常

语法结构：

try:

… 可能出现异常的语句块 …

except 捕获异常的名称:

… 处理异常的语句块 …

finally:

… 异常结束后续执行语句块 …

- Try 块中的语句一旦发生异常，则后续语句将会中止执行，程序控制流直接跳转至 except 块中。
- Exept 若未能捕获到 try 块中的异常，则系统报错
- Finally 块中的语句无论是否出现异常均会执行。
- Try 块必须伴随 except 块同时出现，不能独立出现。

5.4.4 异常的处理 try…except

- 以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

代码演示：ch05-demo11-except01.py

```
try:
    # 不可控，可能用户出错的语句
    choice = int(input('请输入一个整数:>'))
    print('您输入了数字：',choice)
except ValueError:
    # 使用ValueError处理铺货异常后的处理
    print('提示：请输入一个整型数字类型的数据\a')
```

- 首先，执行 try 子句（在关键字 try 和关键字 except 之间的语句）
- 如果没有异常发生，忽略 except 子句，try 子句执行后结束。
- 如果在执行 try 子句的过程中发生了异常，那么 try 子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的 except 子句将被执行。最后执行 try 语句之后的代码。
- 如果一个异常没有与任何的 except 匹配，那么这个异常将会传递给上层的 try 中。

5.4.5 多except捕获

- 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。
- 处理程序将只针对对应的 try 子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。
- 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。

代码演示：ch05-demo12-except02.py

```
try:
    # 不可控，可能用户出错的语句
    num1 = int(input('请输入一个被除数:>'))
    num2 = int(input('请输入一个除数:>'))
    input('%d / %d = %f' %(num1,num2,(num1/num2)))
    continue
except ValueError as err:
    # 使用ValueError处理铺货异常后的处理
    input('提示：请输入一个整型数字类型的数据\a')
    input('错误信息：{0}'.format(err))
    continue
except ZeroDivisionError:
    # 使用ValueError处理铺货异常后的处理
    input('提示：除数不能为零\a')
    continue
except:
    input('系统异常，请联系管理员或拨打客服电话120\a')
    continue
```

- **exept** 异常类名称 **as** 异常对象 ：
- 异常对象可以输出报错的具体信息

5.4.6 try…catch…else/finally

- try except else 语句还有一个可选的 else 子句，如果使用这个子句，那么必须放在所有的 except 子句之后。这个子句将在 try 子句没有发生任何异常的时候执行。例如:

代码演示：ch05-demo13-except03.py

```
try:
    a = 1/0
    print(a)
except ZeroDivisionError:
    print('报错：除数不能为零')
else:
    print('程序结束')
```

- **else** 中的语句当 try 块出现异常时不执行，当 **try** 块没有异常时则执行。
- 同样，若替换成 finally 则无论 try 块中是否出现异常，finally 中的语句块均执行。

5.4.7 自定义条件手动抛出异常

- 在 Python 语言中，并不能解决开发中所有的错误情况。
- 因此，Python3 为我们提供了 raise 关键字，可以根据我们自身业务的特殊情况，自定义错误条件并手动抛出异常。

代码演示：ch05-demo14-raise.py

```
try:
    age = int(input('请输入年龄:>'))
    # 自定义错误条件
    if age < 18:
        # 使用raise抛出异常
        raise NameError('年龄不能小于18岁')
except NameError as err:
    print('报错：{0}'.format(err))
finally:
    print('程序结束')
```

- 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。

5.5 本章总结

本章通过对 for 循环的介绍，再次了解循环体系和使用方法。同时本章介绍的推导式时 Python 编程语言的特色和重点，可以帮助我们快速完成简单的循环迭代处理。最后介绍异常处理机制，通过对各种异常处理的技术介绍让大家对软件编程中的异常处理有一个较为清晰的认知。

第06章：Python函数编程

知识点

目标 1：函数的基本概念及应用

目标 2：带参函数和不带参函数

目标 3：带返回值函数及多值返回函数

目标 4：函数的参数

技能点

实战任务 1：函数编程模式实现用户登录

实战任务 2：利用可变长度参数实现学生成绩录入

6.1 函数编程

6.1.1 函数基础知识

掌握自定义函数的基本语法规范和调用方法 / 掌握函数的各种参数的使用及调用规则

6.1.2 Python函数

- 函数（*Function*）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。
- 我们已经接触过 Python 提供的许多内建函数，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**。

6.1.3 函数基本语法规则

- 函数（*Function*）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。
- 我们已经接触过 Python 提供的许多内建函数，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**。

6.1.4 自定义一个函数语法

- **定义函数的语法：**

  def 函数标识名称(参数列表):

  "函数_文档字符串，对函数进行说明"

  函数体

  return *[表达式]*

- 默认情况下，参数值和参数名称是按函数声明中定义的顺序匹配起来的。

示例（ch7-demo01-function-basic.py） ：

以下为一个简单的 Python 函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

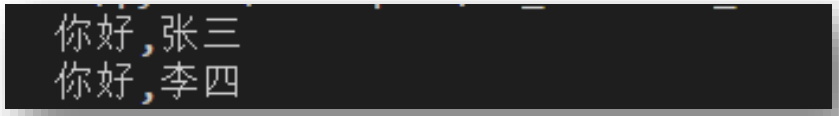
```
'''
    @name: printme
    @args: str
    @return: none
    @date: 2018-04-16
'''
def printme(nickname):
    ''' 输出用户的昵称 '''
    # 输出
    print('你好,{0}'.format(nickname))
    return None
```

6.1.5 函数的调用

- 定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。
- 这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从 Python 提示符执行。
- 如下实例调用了 printme（）函数：

```
# 调用函数
printme('张三')
printme('李四')
```

- 调用后输出的结果为：



6.1.6 return关键字

- **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**。
- 之前的例子都没有示范如何返回数值，下例告诉你怎么做（*demo10.py*） ：

```
#-*- coding: utf-8 -*-
__author__ = "中软国际教育科技·CTO 办公室"
# 定义一个函数
def sum(arg1, arg2):
    '返回两个参数之和'
    total = arg1 + arg2
    # 返回
    return total
# 调用函数
total = sum(10, 20)
print 'total', total

total 30
```

6.1.7 参数的传递

- 在 python 中，类型属于对象，变量是没有类型的：

  a=[1,2,3]

  a="Runoob"

- 以上代码中，[1,2,3] 是 **List** 类型，“Runoob” 是 **String** 类型，而变量 a 是没有类型，它仅仅是一个对象的引用（一个指针），可以是 List 类型对象，也可以指向 String 类型对象

6.2.1 可变（mutable）和不可变（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：**在程序编程中的值传递，如 整数、字符串、元组。如 fun（a），传递的只是 a 的值，没有影响 a 对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。我们经常称之为 **值传递**
- **可变类型：**类似编程中的**引用传递（址传递）**，如 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。

示例（ch7-demo02-args-mutable.py）：

- **python 传不可变对象实例**

```python
'''
    @name: changeString
    @args: str
    @date: 2018-04-16
'''
def changeString(words):
    ''' 尝试修改字符串的值 '''
    # 将words重新赋值
    words = 'home'
    # 输出修改后的结果
    print('在函数体内修改后的结果为:>{0}'.format(words))
    pass

# 调用函数
words = 'school'
changeString(words)
print('在函数体外输出words的值:>{0}'.format(words))
```

代码说明：

函数 changeString 接受一个参数为字符串类型，并在函数体内将传入参数 words 的值重新修改为 home 并输出。在调用后，发现 words 的值并没有进行修改。

再次强调，不可变类型作为函数参数的时候，相当于操作的是**不同的**变量。

自定义函数语法

python 传可变对象实例

示例（ch0-demo03-args-immutable.py）：

```python
'''
    @name: changeList
    @args: str
    @date: 2018-04-16
'''
def changeList(myList):
    ''' 尝试修改List列表对象中的第1个元素值 '''
    # 为myList中的第一个参数赋值
    myList[0] = 'A'
    # 输出修改后的结果
    print('在函数体内修改后的结果为:>{0}'.format(myList))
    pass

# 调用函数
myList = ['a', 'b', 'c']
changeList(myList)
print('在函数体外输出words的值:>{0}'.format(myList))
```

代码说明：

函数 changeString 中的参数为列表（可变类型）对象。因此函数体内修改对象元素时，函数体外的元素同时也被修改。

与不可变参数不同，可变类型作为函数参数的时候，相当于操作的是**相同的**对象。

6.2 函数参数类型

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

```python
'''
    @name: greeting
    @args: str
    @date: 2018-04-16
'''
def greeting(nickname):
    ''' 打招呼 '''
    # 判断用户输入的昵称
    if nickname == 'monster':
        print('我不和怪物说话……')
    else:
        # 输出修改后的结果
        print('你好,{0}'.format(nickname))
    pass

# 调用函数
greeting()
```

```
Traceback (most recent call last):
  File "d:/dev/python/workspace/M1_Course01_Demos/CH07_Demos/ch07-demo04-args-necessary.py", line
 28, in <module>
    greeting()
TypeError: greeting() missing 1 required positional argument: 'nickname'
```

6.2.1 关键字参数

- 关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。
- 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，因为 Python 解释器能够用参数名匹配参数值。
- **示例**（ch07-demo05-keyword.py），以下实例在函数 **printinfo()** 调用时使用参数名：

```
'''
    @name: printinfo
    @args: str
    @date: 2018-04-16
'''
def printinfo(name, age):
    ''' 输出用户的姓名和年龄 '''
    # 输出信息数据
    print('姓名：' + name)
    print('年龄：', age)
    pass

# 调用函数
printinfo(age = 18, name = '卫斯理')
```

```
姓名：卫斯理
年龄： 18
```

6.2.2 必备参数

- 必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。
- **示例**（ch07-demo04-args-necessary.py）调用 **greeting()** 函数，你必须传入一个参数，不然会出现语法错误：

6.2.3. 缺省参数

- 调用函数时，缺省参数的值如果没有传入，则被认为是**默认值**。
- **示例**（ch07-demo06-args-default.py），打印默认的 age，如果 age 没有被传入：

```
'''
    @name: register
    @args: str, int, str, str
    @date: 2018-04-16
'''
# 创建一个空的字典用于存储注册用户信息
person = {}
def register(nickname, age, sex='男', city='北京'):
    ''' 获取用户基本信息 '''
    # 输出信息数据
    person['nickname'] = nickname
    person['age'] = age
    person['sex'] = sex
    person['city'] = city
    pass

# 调用函数
register('张三', 18)
# 输出accounts
print(person)
```

```
{'nickname': '张三', 'age': 18, 'sex': '男', 'city': '北京'}
```

注意：缺省值必须放在最后一个参数

6.2.4 不定长参数 *args和**kw

- 可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做**不定长参数**
- 适用于当参数个数不确定或根据调用情况其参数个数会动态变化的情况。
- **基本语法如下：**

  def 函数名称(formal_args, *args):

  ″函数_文档字符串″

  函数体

  return [*表达式*]

  加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可，**可变长参数的类型为元组**。

示例（ch07-demo07-args-varlen.py），可变长参数：

```python
'''
    @name: register
    @args: str, *args
    @date: 2018-04-16
'''
def employeeInfo(name, *skill):
    ''' 获取用户基本信息 '''
    # 输出信息数据
    print('员工姓名: %s' % name)
    # 掌握技能
    print('专业技能: {0}'.format(skill))
    print('~' * 20)
    pass

# 调用函数
employeeInfo('王晓')
employeeInfo('张艳',['HTML5', 'Java'])
employeeInfo('大壮', 'Python', 'Hadoop', 'HBase')
```

```
员工姓名: 王晓
专业技能: ()
~~~~~~~~~~~~~~~~~~~~
员工姓名: 张艳
专业技能: (['HTML5', 'Java'],)
~~~~~~~~~~~~~~~~~~~~
员工姓名: 大壮
专业技能: ('Python', 'Hadoop', 'HBase')
```

*argv 接受的是一个元组参数

不定长参数

补充 ：kw**

- ** 两个型号代表接受的是一个可变长度的 **字典** 类型的参数。
- 因此，改参数必须以 **k-v** 值结构出现。

def 函数名称(formal_args, **kw):

″函数_文档字符串″

函数体

return [*表达式*]

加了星号（**）的变量名会存放所有未命名的变量参数。选择不多传参数也可，**可变长参数的类型为字典**。

示例（ch07-demo08-args-dict.py），可变长参数：

```
'''
    @name: student
    @args: str, **kw
    @date: 2018-04-16
'''
def student(name, **kw):
    ''' 获取学生基本信息 '''
    # 输出信息数据
    print('学生姓名: %s' % name)
    # 附加信息
    print('考试成绩: {0}'.format(kw))
    print('~' * 20)
    pass
```

- **调用方式：**

```
# 调用函数
student('张大龙')
student('周琪', score1=90)
student('任盈盈', score1=70, score2=85, score3=98)
# 也可以这样搞
scores = {'score1':88, 'score2':78, 'score3':90}
student('张兰', **scores)
```

**kw 接受的是一个字典类型参数

总结 ：*argv 和 **kw 的区别

- 两个参数必须为函数定义中参数列表中的排名最后的参数。
- ***argv** 代表该参数位置可以放任意个数的数据，最终都会转换成 元组 数据类型在函

数体内处理。

- **kw 代表该参数位置可以放 k=v 格式的数据，最终都会转换成 字典 类型数据变函数体内处理。

6.2 本章总结

本章主要介绍了 Python 函数编程的基本模式，对自定义函数以及函数的各类参数应用进行了详细的介绍，希望通过本章节的学习，可以掌握 Python 函数编程的基本应用技巧，后续灵活地应用到我们的编程过程中，提高代码的复用度，提高我们的开发效率。

第07章：Python函数编程进阶

知识点

目标 1：匿名函数 lambda 表达式

目标 2：递归调用模式（扩展）

目标 3：高阶函数

目标 4：装饰器

目标 5：生成器（扩展）

技能点

实战任务：递归实现 n!的阶乘

实战任务：递归实现文件夹检索遍历

实战任务：高阶函数实现一元二次方程求解

实战任务：装饰器的应用

7.1 lambda 表达式

掌握 lambda 表达式的作用和应用场景/掌握其各种应用技巧

7.1.1 匿名函数概述

- 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 *私有参数列表之外* 或 *全局命名空间里的参数*。
- lambda 函数看起来好像只能写一行，其真正的目的是为了简化代码。

- **语法**：lambda 函数的语法只包含一个语句-->
- lambda [arg1 [,arg2,.....argn]] : 表达式

匿名函数 lambda 语法

- **语法：**lambda 函数的语法只包含一个语句

```
Lambad表达式对象 = lambda [arg1 [,arg2,.....argn]] : 表达式
```

- 内置函数
- 自定义函数
- 运算表达式
- 推导式

- **示例：**计算两个整数的和（ch08-demo01-lambda-basic.py）：

```python
# 定义一个lambda表达式实现两个数字的和
lambdaSum = lambda x, y : x + y
# 调用
res = lambdaSum(10, 20)
# 输出
print('num1+num2=%d' % res)
```

=

```python
# 定义一个函数实现连个数字的和
def funSum(num1, num2):
    return num1 + num2
# 调用函数
res = funSum(10, 20)
# 输出
print('普通函数实现:> num1+num2=%d' % res)
```

7.1.2 lambda的表达式应用

示例1：使用内置函数作为表达式

- ch08-demo02-lambda-advance.py

```python
# lambda表达式调用build-in函数
lamExp1 = lambda x : print(x)
# 调用
lamExp1(3)
```

示例2：使用自定义函数作为表达式

- ch08-demo02-lambda-advance.py

```python
# lambda表达式调用自定义函数
import time
def formatTime(nowTime):
    return time.strftime('%Y-%m-%d')
lamExp2 = lambda timedate : formatTime(timedate)
# 调用
strTime = lamExp2(time.localtime())
# 输出
print('当前日期:> %s' % strTime)
```

示例3：使用推导式作为表达式

- ch08-demo02-lambda-advance.py

```python
# lambda表达式调用推导式
lamExp3 = lambda x : [i**2 for i in range(x)]
# 调用
list1 = lamExp3(3)
# 输出
print('结果为:> {0}'.format(list1))
```

示例4：使用默认值参数

- ch08-demo03-lambada-params.py

```python
# lambda表达式携带缺省值参数（默认值参数）
lamExp1 = lambda x, y=10 : x + y
print(lamExp1(2))
```

示例5：使用不定长参数 *args

- ch08-demo03-lambada-params.py

```python
# lambda表达式携带可变长参数1
lamExp1 = lambda *args : [i**2 for i in args]
list1 = lamExp1(1,2,3)
print(list1)
```

示例6：使用不定长参数 **kw

- ch08-demo03-lambada-params.py

```python
# lambda表达式携带可变长参数2
lamExp1 = lambda **kw : {item for item in kw.items()}
dict1 = {'a':'A', 'b':'B'}
dict2 = lamExp1(**dict1)
print(dict2)
```

7.2 变量作用域

- 一个程序的所有的变量并不是在哪个位置都可以访问的。
- **访问权限**决定于这个变量是在哪里赋值的。
- 变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。

两种最基本的变量作用域如下：

① 全局变量

② 局部变量

7.2.1 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**，定义在函数外的拥有**全局作用域**。
- **局部变量**只能在其被声明的函数内部访问
- **全局变量**可以在整个程序范围内访问。

示例（ ch08-demo05-variable-scope01.py），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

```
# 定义一个全局变量total
total = 10

# 定义一个函数
def calcSum(num1, num2):
    '返回两个数的和'
    # 局部变量total
    total = num1 + num2
    # 输出
    print('函数内的total:> %d' % total)
    # 返回
    return total

# 调用函数
calcSum(10, 20)
# 函数外部
print('函数外部访问变量total:> %d' % total)
```

- 全局变量想作用于函数内，需加 **global**
- **示例**（ ch08-demo05-variable-scope02.py ）：

```python
# 全局变量
globalvar = 0

# 定义一个函数
def set_globalvar_one():
    # 在函数内声明全局变量的内部作用域使用权
    global globalvar
    # 为全局变量赋值
    globalvar = 1

# 定义一个函数
def print_globalvar():
    # 先查找函数内是否有局部globalvar变量?
    # 没有再找全局，再没有报错
    print('print_globalvar函数中的输出:> %d' %globalvar)
```

```python
# 脚本程序入口
if __name__ == '__main__':
    print('函数体外的输出:> %d' % globalvar)  # 输出全局变量
    # 调用set_globalvar_one()函数
    set_globalvar_one()
    # 调用pint_globalvar()函数
    print_globalvar()
```

1、global---将变量定义为全局变量。可以通过定义为全局变量，实现在函数内部改变变量值。

2、 一个 global 语句可以同时定义多个变量，如 global x, y, z。

7.2.2 递归 Recursion

掌握了解递归的基本工作模式和应用方法

递归的实现

- 递归：在调用一个函数的过程中，直接或间接地调用了函数本身这个就叫递归

直接调用自身函数的示例（ ch08-demo06-recursion.py ）：

```python
# 方式1：自身调用自己
def func():
    print('from func')
    time.sleep(1)
    func() # 在函数体内自己调用自身
    pass
```

间接调用自身函数的示例（ ch08-demo06-recursion.py ）：

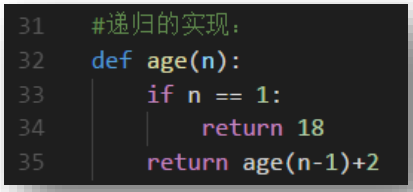
```python
# 方式2：简介调用自己
def foo():
    print('from foo')
    time.sleep(1)
    bar()

def bar():
    print('from bar')
    time.sleep(1)
    foo()
```

问题：

- 能告诉我一下代码运行的结果吗？

示例（ ch08-demo06-recursion.py ）：



```
#递归的实现：
def age(n):
    if n == 1:
        return 18
    return age(n-1)+2
```

- **递归运行说明：**

  age(5)=age(4)+2 第一次进入

  age(4)=age(3)+2 第二次进入

  age(3)=age(2)+2 第三次进入

  age(2)=age(1)+2 第四次进入

  age(1)=18 第五次进入，最后判断终止条件

7.2.3 示例1：递归实现阶乘

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 n! = 1 * 2 * 3 * ... * n，用函数 factorial(n)表示，可以看出：

factorial(n) = n! = 1 * 2 * 3 * ... * (n-1) * n = (n-1)! * n = fact(n-1) * n

所以，factorial(n)可以表示为 n * factorial(n-1)，只有 n=1 时需要特殊处理。

于是，factorial(n)用递归的方式写出来就是：

- **示例**（ ch08-demo07-recursion-factorial.py ）：

```python
# 创建一个递归函数
def factorial(n):
    if n==1:
        return 1
    return n * factorial(n - 1)

# 脚本程序入口
if __name__ == '__main__':
    # 调用函数
    res = factorial(5)
    print('阶乘的结果为:> %d' %res)
```

7.2.4 示例2：递归实现遍历指定文件或磁盘的检索

前置知识补充：

- os.listdir(filePath)　返回一个列表对象，存放指定路径下的所有文件夹或文件名称
- os.path.join(filePath, fi)　文件路径拼接
- os.path.isdir(fi_d)　判断当前路径是否为一个文件夹
- os.sep() 系统分隔符

示例（ ch08-demo08-recursion-dirs.py ）：

```python
# 自定义一个函数
def recursionFilePath(filePath):
    """ 递归检索指定磁盘或文件夹结构 """
    # 步骤1：遍历指定的文件夹
    files = os.listdir(filePath)
    # 步骤2：for循环遍历数据集合
    for fi in files:
        # 重点语句：将路径进行拼接申城组合后的绝对路径
        fi_d = os.path.join(filePath, fi)
        # 判断当前循环的对象是否为文件夹
        if os.path.isdir(fi_d):
            # 继续调用该函数进行检索
            recursionFilePath(fi_d)
        else:
            print("==>{0}".format(fi_d))

# 脚本程序入口
if __name__ == '__main__':
    # 调用函数
    recursionFilePath("d:" + os.sep + "dev")
```

7.3 高阶函数

理解什么是高阶函数？以及高阶函数的基本应用方法

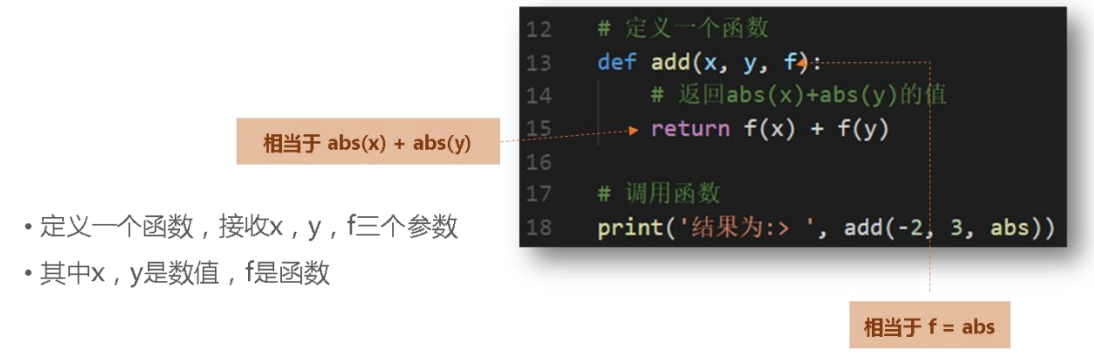
7.3.1 高阶函数定义

能接收函数做参数的函数。

- 变量可以指向函数
- 函数的参数可以接收变量
- 一个函数可以接收另一个函数作为参数

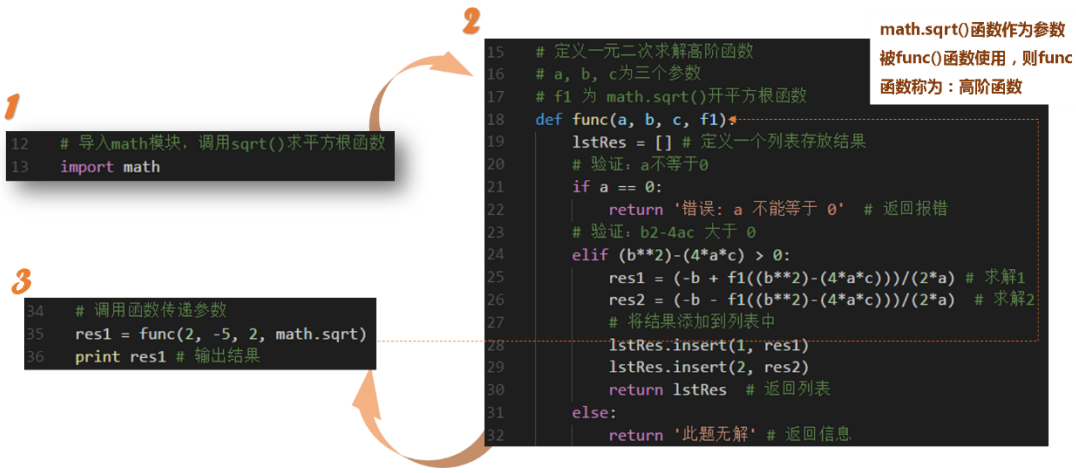
7.3.2 高阶函数的应用

示例（ch08-demo09-advfunc.py）：**接收 abs 函数**



示例（ch08-demo10-equation.py）：**利用高阶函数实现线型代数一元二次方程求解函数**

- 在线型代数中，经典的一元二次方程求解：$ax^2 + bx + c = 0$ (*满足条件：* $a \neq 0$ *并且* $b^2 - 4ac > 0$，*否则无解*)
- 求解公式为：$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$



7.4 装饰器 Decorator

装饰器的应用技巧及应用场景

7.4.1 装饰器的定义

- 装饰模式有很多经典的使用场景，例如插入日志、性能测试、事务处理等等。
- 有了装饰器，就可以提取大量函数中与本身功能无关的类似代码，从而达到代码重用的目的。

7.4.2 装饰器的应用

下面就一步步看看 Python 中的装饰器。

示例：ch08-demo11-decorator.py

```
# 定义一个装饰器函数
def decoratorFunc(f):
    print('前置执行……')
    f() # 调用函数
    print('后置执行……')
    pass

# 装饰器应用
@decoratorFunc
def func1():
    print('hello, 中软国际')
    pass
```

运行结果：

```
前置执行……
hello, 中软国际
后置执行……
```

补充示例：程序执行性能测试（ch08-demo12-decorator-performance.py）

- 在每次调用某个函数的时候，都要计算一下被调用的函数所消耗的时间。

```python
# 导入time模块
import time

# 自定义一个统计时间的高阶函数
def calcTime(func):
    print('--> 开始计时.....')
    startTime = time.time() # 使用time()获取时间戳
    func()  # 调用传入的函数
    endTime = time.time()  # 使用time()再获取时间戳
    print('--> 结束计时.....')
    msecs = (endTime - startTime) * 1000  # 计算两次时间差
    # 输出结果
    print('--> 性能测试结果：' + func.__name__ + '()函数耗时：%f ms' % msecs)
```

- 常规调用方式（非装饰器方式）

```python
# 自定义一个测试函数
def func1():
  print 'func1() start...'
  time.sleep(0.6)  # 模拟 0.6 秒执行
  print 'func1() end...'
# 调用高阶函数
calcTime(func1)
```

- 装饰器绑定指定的自定义函数

```python
# 自定义一个测试函数，装饰器方式，无需调用
@ calcTime
def func1():
    print('func1() start...')
    time.sleep(0.6)  # 模拟0.6秒执行
    print('func1() end...')
```

7.5 闭包 Closure

闭包的基本概念及应用技巧

7.5.1 闭包的定义

闭包 closure

- 在一个外函数中定义了一个内函数，内函数里运用了外函数的临时变量，并且外函数的返回值是内函数的引用。这样就构成了一个闭包。（*相当于给内部函数起别名*）。
- 实现规则：
  ① 定义内部函数
  ② 外部函数必须有返回值，而且返回值必须为内部函数对象

- **闭包结构的定义，示例（ ch08-demo13-closure.py ）：**

```python
#闭包函数的实例
# outer是外部函数 a和b都是外函数的临时变量
def outer(a):
    b = 10
    # inner是内函数
    def inner(c):
        #在内函数中 用到了外函数的临时变量
        print(a+b+c)
    # 外函数的返回值是内函数的引用
    return inner
```

7.5.2 闭包的应用

闭包结构的调用

```python
# 调用外部函数outer,实质是获取了内部函数的引用
demo = outer(5)
print(type(demo))
# 调用内部函数inner
print(demo(3)) # 18
```

扩展：内部函数为匿名函数

- 由于功能逻辑相对简单，为了简化代码，我们可以考虑使用匿名函数（lambda）实现。

- lambda 作为内部函数，示例（ ch08-demo13-closure.py ）：

```python
# 匿名函数作为内部函数
def outerFunc(a):
    b = 10
    return lambda c : a+b+c
```

  - 调用方式

```python
# 调用outerFunc()函数
f1 = outerFunc(5)
res = f1(3) # 调用内部lambda表达式（匿名函数）
print(res)
```

7.7 生成器 Generator

掌握生成器的两种创建方法/掌握 yield 关键字的使用

7.7.1 生成器的定义

生成器 Generator

- 之前，我们在学习 推导式 的时候提到过。
- 回忆：使用( ) 不代表 元组推导式，而是实现了一个生成器，解析器在实时生成数据，数据不会驻留在内存中。

  因此，其执行效率很高！

7.7.2 生成器的应用

- 使用推导式快速创建一个生成器，示例（ ch08-demo14-generator.py ）：

```python
# 推导式方式创建一个生成器Generator
myGenerator = (i**2 for i in range(10) if i%2 == 0)
# 使用for循环遍历元素
for item in myGenerator:
    print(item)
    pass
```

7.7.3 生成器 yield 关键字

- yield 是一个类似 return 的关键字，只是这个函数返回的是个生成器
- 当你调用这个函数的时候，函数内部的代码并不立即执行 ，这个函数只是返回一个生成器对象
- 当你使用 for 进行迭代的时候，函数中的代码才会执行

使用 yield 创建一个生成器，示例（ ch08-demo14-generator.py ）：

```
19  # 使用yeild方式创建一个生成器
20  def createGenerator():
21      for i in range(10):
22          yield i**2 # yield关键字
23      pass
24  myGenerator = createGenerator()
25  # for循环遍历生成器
26  for item in myGenerator:
27      print(item)
28      pass
```

7.8 本章总结

本章通过对 Python 函数的讲解，全面认知和了解函数的定义方法及其作用和应用场景。通过对常用算法递归的介绍以及与实际应用结合，让学习者更快地掌握递归算法的实际应用。最后通过对高阶函数、张诗琪、闭包以及生成器的介绍，全面了解并掌握 Python 编程语言中的高级对象，为后续的学习和开发过程中打下坚实的基础。

第08章：面向对象编程基础

知识点

目标 1：编程模式的变迁

目标 2：面向对象编程基础知识

目标 3：Python 面向对象的快速实现

目标 4：面向对象中的四种方法及变量

技能点

实战任务 1：游戏人生

实战任务 2：磁盘指定类型文件检索

8.1 编程模式的变迁

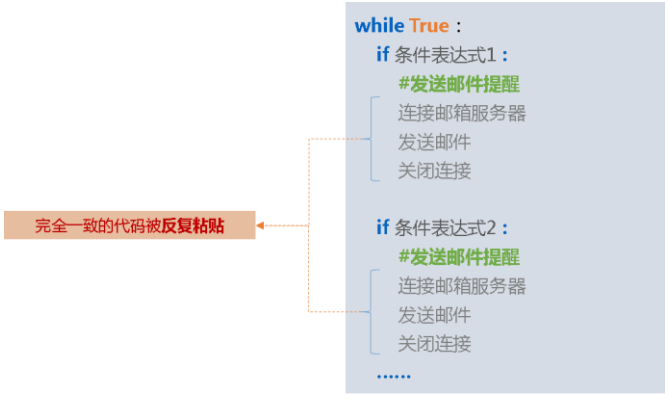
Python 编程模式的发展 / 从面向过程编程 ->函数编程 -> 面向对象编程

8.1.1 概述

- 面向过程：根据业务逻辑从上到下写垒代码
- 函数式：将某功能代码封装到函数中，日后便无需重复编写，仅调用函数即可
- 面向对象：对函数进行分类和封装，让开发“更快更好更强…”

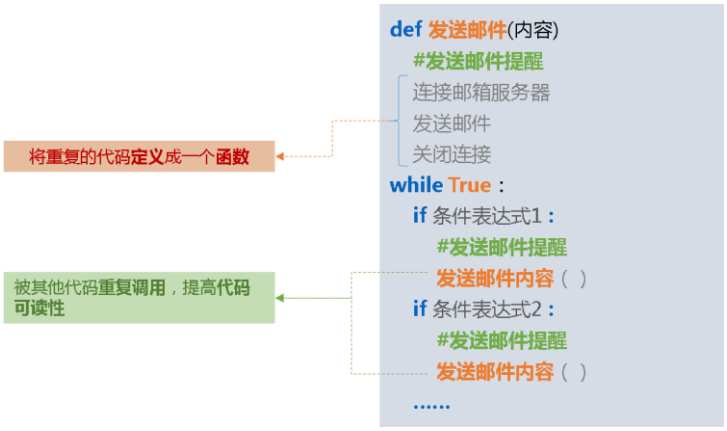
8.1.2 从面向过程开始

面向过程编程最易被初学者接受，其往往用一长段代码来实现指定功能，开发过程中最常见的操作就是粘贴复制，即：将之前实现的代码块复制到现需功能处。



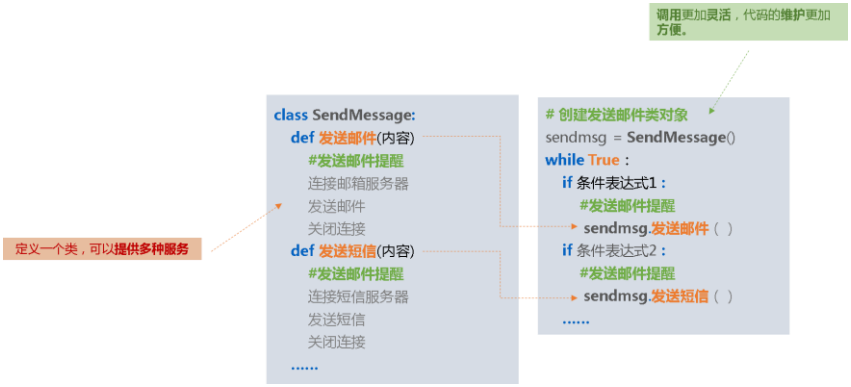
8.1.3 开启函数式编程模式

- 随着时间的推移，开始使用了**函数式编程**，增强代码的重用性和可读性，就变成了这样：



8.1.4 面向对象的编程模式

- 随着业务复杂度的不断提高，面向对象编程模式使得我们开发变得更加灵活。
- **降低代码**之间的**耦合度**，强调**模块开发**的思想。
- 自由**拆分**和**组合**功能，以不同的组合形式体现，从而提供不同的服务。
- 同时**提高**团队协作**开发的效率**。



8.2 面向对象编程基础知识

面向对象的定义 / 抽象的分析方法 / 类及对象的概念 / 面向对象三大特征

8.2.1 面向对象的概念

- **面向对象编程**（Object Oriented Programming，**OOP**，面向对象程序设计）
- 世间万物一切皆是“**对象**”（如：汽车、动物、空气 ……，万物皆对象）
- 需要高度**抽象**的思维方式去体会这个概念。

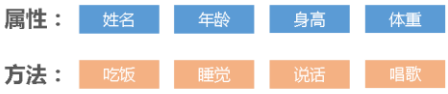
8.2.2 对象的组成（状态+行为）

- **任何对象**都存在着自己独有的**状态**和**行为**。
- **状态：**都是描述事物的**名词形式**。
- **行为：**都是描述事务的**动词形式**。

8.2.3 抽象过程

- 所谓**抽象过程**是**面向对象中**的一种**分析方法。**

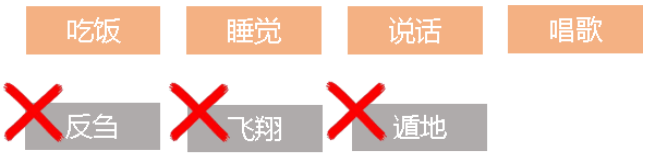
- **抽象过程**就是要分析出对象都存在哪些自己的**属性（状态）**和**方法（行为）**。



8.2.4 类

- **类**就是**类别**的意思，**同一类别的事物都会有 共同的属性（状态）**或 **方法（行为）。**
- 人类拥有**共同的属性（状态）**可以描述：

- 人类也有**共同的方法（行为）**可以描述：



- 正所谓“**物以类聚，人以群分**”，不同类有着**明显**的属性和方法的**区别**

8.2.5 对象

- **对象**就是**类的一个具体表现事物。**
- 任何对象都拥有**相同**的**属性**和**方法**（即类中的所有属性和方法）。
- 每个对象的属性或方法会有独特的**个异性**。

8.2.6 面向对象的三大特征概述

- **继承：**子类拥有父类的可访问属性和方法，**提高代码的重用性。**
- **封装：**通过语句块保护变量或对象的作用域，**提高代码的安全性。**
- **多态：**各种重载和重写的机制，**提高代码灵活性。（特别说明，Python OOP 没有多态特性）**

8.3 Python 面向对象的快速实现

类的创建 / 对象的创建及使用 / 构造方法

8.3.1 如何创建一个类

- class 是关键字，表示类
- 类创建的**语法：**

```
class 类名称:
    ..... 变量 .....
    ..... 方法 .....
```

示例： ch09-demo01-class.py

特别说明：Python类中定义的函数称为**方法**

```python
# -*- coding:utf-8 -*-
'''
    ch09-demo01-class.py
    ----------------------
    类和对象的基本应用

    @Copyright: Chinasoft International. ETC
    @Author: alvin
    @date: 2018-04-18
'''

# 创建一个类
class Person():

    # 类成员方法
    def greeting(self):
        print('你好')
        pass
    pass
```

8.3.2 如何创建一个对象

- 对象创建的**语法**：

```
对象标识符 = 类名称 ( )
```

- 对象使用访问操作符 “.” 对类的方法进行调用访问。

```
对象标识符.方法名称 ( )
```

```python
# 脚本程序入口
if __name__ == '__main__':
    # 使用Person类创建一个对象1
    obj1 = Person()
    # 访问调用obj1对象中的greeting()成员方法
    obj1.greeting()

    # 使用Person类创建一个对象2
    obj2 = Person()
    # 访问调用obj1对象中的greeting(
    obj2.greeting()
```

```
你好
你好
```

- 创建对象的过程也称之为：**类实例化过程**。代码见（ch09-demo01-class.py）

8.4 类的方法

- **Python3 面向对象的概念中，关于类的方法一共有四种：**
  - ① 实例方法 （ instanceMethod ）- 使用频次最多
  - ② 类方法（classMethod）- 根据需求适度
  - ③ 静态方法（staticMethod）- 根据需求适度
  - ④ 普通方法（ordinaryMethod）- 使用频次很少
- **四种方法的区别：**
  - ① 定义语法格式上；
  - ② 对各种类型变量的访问权限上。

8.4.1 实例方法 instanceMethod

- **实例方法**是属于实例对象的，但也可以使用类直接调用。但两者调用的方式有区别（*传递参数个数上面*）。
- **实例方法**在定义中，第一个参数必须为 **self** （*当然参数名称也可以随意定义，但是 self 是 python 默认的规定最好不修改*）
- **实例方法**定义的**语法规范：**

  def 方法名称 (self [, arg1[, arg2, …[, argN]]]) :

实例方法的定义及各种调用

- 第一个参数 **self** 代表 调用该方法的实例对象，在使用实例对象调用方法的时候 可以忽略第一个参数，只需传递满足在实例方法定义过程中的 **n-1** 个后续参数数量即可。
- 而类调用时，需要传递与定义数量匹配的 **n** 个参数，第一个 self 只当做一个参数变量名称而已。

示例：ch09-demo02-instancemethod.py

```python
# 创建一个类
class ClassName():

    # 定义一个类的实例方法
    def instanceMethod(self, arg1, arg2):
        print('>> {0}'.format(self))
        print('>> {0}'.format(arg1))
        print('>> {0}'.format(arg2))
        pass
    pass
```

```python
# 脚本程序入口
if __name__ == '__main__':
    # 使用Person类创建一个实例对象
    instanceObj = ClassName()
    # 使用类实例对象调用instanceMethod()实例方法
    instanceObj.instanceMethod(10, 20)
    print('-' * 30)
    # 使用类直接调用instanceMethod()实例方法
    ClassName.instanceMethod(-10, 30, 40)
```

实例对象调用实例方法

类直接调用实例方法

8.4.2 特殊的类实例方法1-构造方法

—— 构造方法 __init__

- 构造方法是类成员方法中**特殊的一种方法，其方法名称为 __init__( )**。
- 该方法无需对象通过访问操作符调用，在类实例化对象的过程中自动调用。
- 构造方法的作用有**两个**：
  - ① 为对象创建内存空间
  - ② 实例对象的实例变量参数初始化

语法：

```
class 类名称:
    # 类的构造方法
    def __init__(self, 参数列表 ):
        ……方法体……
```

示例：ch09-demo03-constructor.py

```
# 创建一个类
class Person():

    # 类的构造方法（特殊的实例方法）
    def __init__(self):
        print('Person类的构造方法……')
        pass
    pass

# 脚本程序入口
if __name__ == '__main__':
    # 使用Person类创建一个对象1
    instanceObj = Person()
```

定义类构造方法

实例化类对象，自动调用构造方法

8.4.3 特殊的类实例方法2-实例对象输出

对象输出方法 __str__

- 当我们使用 print() 函数输出使用类创建好的实例对象的时候，默认输出的是 当前实例对象的十六进制内存地址。

- **示例**：ch09-demo04-tostring.py

```
# 脚本程序入口
if __name__ == '__main__':
    # 使用Person类创建一个对象1
    instanceObj = Person()
    # 使用print()直接输出对象
    print('>> {0}'.format(instanceObj) )
```

```
>> Person类的构造方法.....
>> <__main__.Person object at 0x00000242C7A9BD68>
```

- 改造，重新定义对象实例的输出。在类中定义 __str__方法

```
    # 定义类实例对象的输出
    def __str__(self):
        return '自定义设置Person类对象输出内容.....'
```

```
>> Person类的构造方法.....
>> 自定义设置Person类对象输出内容.....
```

8.4.4 实例变量（成员属性）

- 与**实例方法** 遥相呼应 的是 实例变量。
- 实例变量 属于类创建的实例对象的，这些属性是所有实例对象都拥有的。只不过每个实例对象的属性值各异。
- 实例变量 均在构造方法__init__( )中 声明并初始化。
- **语法：**

  self.*实例变量名称* = 值　　# *self 代表当前创建的实例对象*

- 类中的实例方法均可以访问实例变量。

示例 1：实例变量的创建及初始化

- 一般在创建构造方法的时候，充分利用构造方法为类实例变量（成员属性）初始化的这一典型作用。

示例：ch09-demo5-instancevar.py

```
# 创建一个类
class Person():

    # 类的构造方法
    def __init__(self, name, age):
        # 定义类的实例变量
        self.name = name
        self.age = age
        pass
    pass
```

在构造方法中，**创建实例变量并初始化**

- self 是对类创建对象的引用，代表**当前对象**。

```
# 脚本程序入口
if __name__ == '__main__':
    # 使用Person类创建一个对象1
    # 自动执行Person类的构造方法并传递参数
    obj1 = Person('路人甲', 16)

    # 使用Person类创建一个对象2
    # 自动执行Person类的构造方法并传递参数
    obj2 = Person('路人乙', 22)
```

通过**构造方法传参**调用方式，实现对当前类实例对象中的**实例变量初始化**

创建另一个实例对象

- 当执行 obj1 = Person ('路人甲', 16) 时，self 等于 obj1
  当执行 obj2 = Person ('路人乙', 22) 时，self 等于 obj2

示例 2：类的实例变量的两种输出方式

- **示例 2-2**：通过 定义类的实例方法 实现对类创建的对象实例变量进行输出 （ch09-demo7-instancevar-output02.py）
- 由于类中所有的**实例方法**均可以**访问**法类的**实例变量**，则自定义实例方法进行输出。

8.4.5 实战任务：游戏人生

使用面向对象的封装特性完成

业务需求：

- 创建一个游戏场景，涉及 3 个场景和 1 个功能。
  - ① 场景 1：草场作战，消耗战斗力 200
  - ② 场景 2：修炼，增长战斗力 100
  - ③ 场景 3：多人团战，消耗战斗力 500
  - ④ 功能 1：显示玩家状态
- 当玩家战斗力**为 0** 时，阵亡！

技术需求：

- 使用面向对象的封装特性实现任务的创建及场景功能的调用。

Step1：创建人物类

- 创建角色类 Role，并设置构造方法传参（姓名、性别、战斗力）

```
15  # 定义一个角色类
16  class Role():
17
18      # 定义类构造方法
19      def __init__(self, name, sex, fighting):
20          # 为类成员属性赋值
21          self.name = name
22          self.sex = sex
23          self.fighting = fighting
24          pass
```

Step2：创建场景 1 函数

- 在类中创建 3 个游戏场景以及显示状态的函数

```
26      # 定义场景1：草场
27      def glassland(self):
28          '草场作战消耗200战斗力'
29          print('[%s] 参加草场作战.....' % self.name)
30          self.fighting -= 200 # 消耗200战斗力
31          # 判断是否死亡
32          if self.fighting > 0 :
33              print('[%s] 所剩战斗力%d'% (self.name, self.fighting))
34          else:
35              print('[%s] 阵亡.....' % self.name)
36              os._exist(0) # 退出程序
37          pass
```

Step3：创建场景 2 函数

- 创建场景 2：修炼

```
# 定义场景2：修炼
def practice(self):
    '自我修炼增长100战斗力'
    print('[%s] 闭关修炼……' % self.name)
    self.fighting = self.fighting + 100  # 战斗力+100
    print('[%s] 增长战斗力%d' % (self.name, self.fighting))
```

Step4：创建场景 3 函数

- 创建场景 3：多人团转

```
# 定义场景3：多人团战
def incest(self):
    '多人团战消耗500战斗力'
    print('[%s] 参加多人团战……' % self.name)
    self.fighting -= 500  # 战斗力-500
    # 判断是否阵亡
    if self.fighting > 0 :
        print('[%s] 所剩战斗力%d' % (self.name, self.fighting))
    else:
        print('[%s] 阵亡！！！！' % self.name)
        os._exit(1)
```

Step5：创建场景 3 函数

- 创建显示玩家状态的函数

```
58      # 定义函数4：输出个人状态
59      def detail(self):
60          '显示个人状态数值'
61          print('------- [%s] -------' % self.name)
62          print('姓名：%s\n性别：%s\n战斗力：%d' \
63                % (self.name, self.sex, self.fighting))
64          print('-' * 30)
```

Step6：创建 main 入口调用

- **完整案例见** ch09-demo08-example-games.py

```python
# 脚本程序入口
if __name__ == '__main__':
    os.system('cls') # 清屏操作
    print('#' * 30)
    print('人物创建')
    print('#' * 30)
    # 创建角色
    obj = Role('独孤求败', '女', 1000)
    # 输出角色状态
    obj.detail()
    print('#' * 30)
    print('[%s]重出江湖' % obj.name)
    print('#' * 30)
    obj.glassland()    # 进入场景1草场
    obj.incest()       # 进入场景3多人团战
    obj.glassland()    # 进入场景1草场
    obj.practice()     # 进入场景2修炼
    obj.incest()       # 进入场景3多人团战
```

说到这里我们有个疑问？

Q：使用函数式编程和面向对象编程方式来执行一个“方法”时，函数要比面向对象简便？

- 面向对象：【创建对象】->【通过对象执行方法】
- 函数编程：【执行函数】

观察上述对比答案则是肯定的，然后并非绝对，场景的不同适合其的编程方式也不同。

总结：函数式的应用场景 -> 各个函数之间是独立且无共用的数据

8.4.6 类方法 classmethod

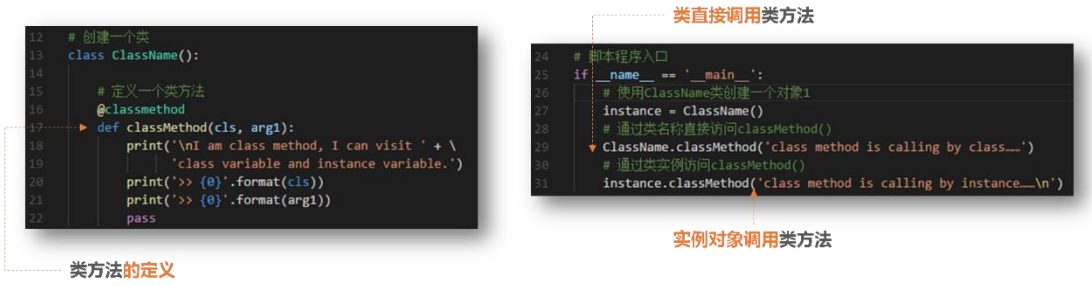
- **类方法**是属于类本身的，但也可以使用对象实例调用。但两者调用的方式有区别（*传递参数个数上面*）。
- **类方法**在定义中，第一个参数必须为 cls （*当然参数名称也可以随意定义，但是 cls 是 python 默认的规定最好不修改*）
- **类方法**定义的**语法规范**：

```
@classmethod
def 方法名称 (cls [, arg1[, arg2, ...[, argN]]]) :
```

使用装饰器方式定义，告知Pyhton解析器该方法为类方法，解析器优化该方法的执行效率。

类方法的定义及各种调用

- 第一个参数 cls 代表 当前的类，在使用类直接调用方法的时候 忽略第一个参数，只需传递满足在实例方法定义过程中的 n-1 个后续参数数量即可。
- 而类的对象实例调用时，同样需要传递与定义数量匹配的 n-1 个参数。
- **示例：**ch09-demo09-classmethod.py




```python
# 创建一个类
class ClassName():

    # 定义一个类方法
    @classmethod
    def classMethod(cls, arg1):
        print('\nI am class method, I can visit ' + \
              'class variable and instance variable.')
        print('>> {0}'.format(cls))
        print('>> {0}'.format(arg1))
        pass

# 脚本程序入口
if __name__ == '__main__':
    # 使用ClassName类创建一个对象1
    instance = ClassName()
    # 通过类名称直接访问classMethod()
    ClassName.classMethod('class method is calling by class……')
    # 通过类实例访问classMethod()
    instance.classMethod('class method is calling by instance……\n')
```

8.4.7 类变量的定义及访问操作

- 类变量 与 类方法 可以成对使用，同样类方法也可访问操作实例变量。
- 类变量 声明在类中，类中的各种类型方法（*除了普通方法之外*）均可以访问。
- **示例：**ch09-demo10-classvar.py


```python
# 创建一个类
class ClassName():

    # 定义一个类变量
    classVariable = 'I am a class Variable.'

    # 定义一个类方法
    @classmethod
    def classMethod(cls, parameter):
        print('\nI am class method, I can visit class variable and instance variable.')

        # 访问类变量
        print('I am class method, I can modifying the class variable')
        classVariable = parameter + ': class variable had updated.'
        print(classVariable)
        pass
```

类方法也可以访问操作实例变量

- 类方法 属于类，因此类方法也可以直接访问实例变量。
- **示例：**ch09-demo10-classvar.py

8.5 静态方法 staticmethod

8.5.1 静态方法的定义

- **静态方法**是属于类本身的，但也可以使用对象实例调用。但两者调用的方式有区别（*传递参数个数上面*）。
- **类方法**在定义中，第一个参数必须为 cls （*当然参数名称也可以随意定义，但是 cls 是 python 默认的规定最好不修改*）
- **类方法**定义的**语法规范**：

8.5.2 静态方法的定义及各种调用

- **静态方法**在调用的时候，需要传递与定义同等个数的参数。
- **静态方法**可以被类或对象实例调用。
- **示例：**ch09-demo11-staticmethod.py

Python3 中没有实际意义的静态变量

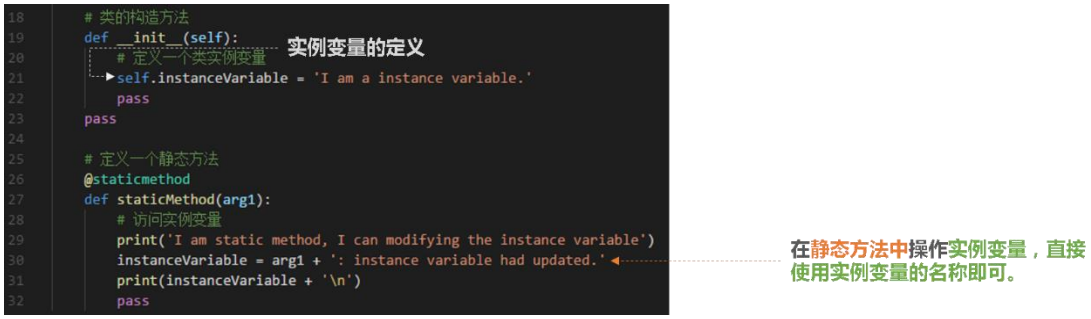
- 由于 Python 编程语言为动态语言，因此严格意义上的静态变是没有必要的。
- 但是，静态方法可以访问 实例变量 和 类变量。
- **示例，静态方法访问类变量：**ch09-demo11-staticmethod.py


```
# 创建一个类
class ClassName():
    # 定义一个类变量
    classVariable = 'I am a class Variable.'

    # 定义一个静态方法
    @staticmethod
    def staticMethod(arg1):
        # 访问类变量
        print('I am static method, I can modifying the class variable')
        classVariable = arg1 + ': class variable had updated.'
        print(classVariable)
        pass
```

8.5.3 静态方法可以访问实例变量

- 类方法 属于类，因此类方法也可以直接访问实例变量。
- **示例，静态方法访问实例变量：**ch09-demo11-staticmethod.py




```
    # 类的构造方法
    def __init__(self):
        # 定义一个类实例变量
        self.instanceVariable = 'I am a instance variable.'
        pass
    pass

    # 定义一个静态方法
    @staticmethod
    def staticMethod(arg1):
        # 访问实例变量
        print('I am static method, I can modifying the instance variable')
        instanceVariable = arg1 + ': instance variable had updated.'
        print(instanceVariable + '\n')
        pass
```

3.6 8.5.4 普通方法 ordinaryMethod

- **普通方法**在类中的定义与定义一个函数无任何区别。
- 普通方法 可以被 类直接调用 但 不能被 对象实例调用。
- **普通方法**无法访问 实例变量 和 类变量。
- **类方法**定义的**语法规范：**

    def 方法名称（[, arg1[, arg2, …[, argN]]]）：

四种方法之间的区别

- **调用方式角度**

	实例方法	类方法	静态方法	普通方法
a = A()	a.foo(x)	a.class_foo(x)	a.static_foo(x)	不可用
A	不可用	A.clas_foo(x)	A.static_foo(x)	A.foo(x)

- **变量访问**

	实例方法	类方法	静态方法	普通方法
实例变量	可以	可以	可以	不可以
类变量	可以	可以	可以	不可以

8.6 本章总结

通过本章的学习，了解和掌握了面向对象的基本概念以及抽象的方法。通过对基本类的定义以及对象的操作，掌握了面向对象在 Python 语言中的基本操作方法。对构造方法、实例输出方法以及实例变量、类变量和静态方法等概念的学习和应用，让学习者快速掌握面向对象的编程思想以及在 Python 中实现的方法。

第09章：Python面向对象编程进阶

知识点

目标 1：封装的概念及表现形式

目标 2：对象的访问权限管理

目标 3：继承的实现方法

目标 4：super 的使用

目标 5：“鸭子类型”的使用

技能点

实战任务 1：封装访问修饰符的应用

实战任务 2：继承的实现方法

实战任务:3：super 的使用

9.1 OOP 三大特征之一：封装

封装技术的优势/ 访问修饰符号 / 变量的访问控制 / 方法的访问控制

9.1.1 封装的概念

- 为什么需要使用封装技术？

  为了保障数据的安全性，降低代码的耦合度。
- 如何定义封装？

  将具有统一功能或相关的代码块进行高度抽象的处理过程。
- 封装的具体表现形式？

  其主要的表现形式就是将一段代码块高度抽象成一个函数、一个类或类中的方法。

9.1.2 访问修饰符

在之前学过的面向对象编程中，我们发现无论是在类中定义的变量还是方法，基本上都可以通过实例对象或类本身进行快速访问调用。

- 数据访问安全问题？

  我们在设计类的时候，有些变量或方法不希望让类外部进行调用，其仅供类内部使用。这

样可以在一定程度上保护数据，以免受到外部误操作的意外篡改，导致数据不一致。

- 访问修饰符？

为了更好地保护数据，Python 提供了三种访问权限级别： 公有、私有 和 受保护

① 公有修饰的变量和方法，类的内部或外部均可调用访问。即为大家公共的数据。

② 私有修饰的变量和方法，仅供类内部使用，本类外部无法访问调用。即不开放数据。

③ 受保护修饰的变量和方法，在程序中只有本类及其子类可以调用。即受限的数据。

9.1.3 访问修饰符的使用

Python 编程语言很巧妙地使用“下划线+标识符”的命名规则，实现了对变量和方法的访问控制。

① 标识符开头无下划线，该变量或方法为 公有权限（即类内外均可访问调用）

self.name **or** **def** show(self)

② 标识符开头双下划线，该变量或方法为 私有权限（即仅类内部自己可以访问调用）

self.__name **or** **def** __show(self)

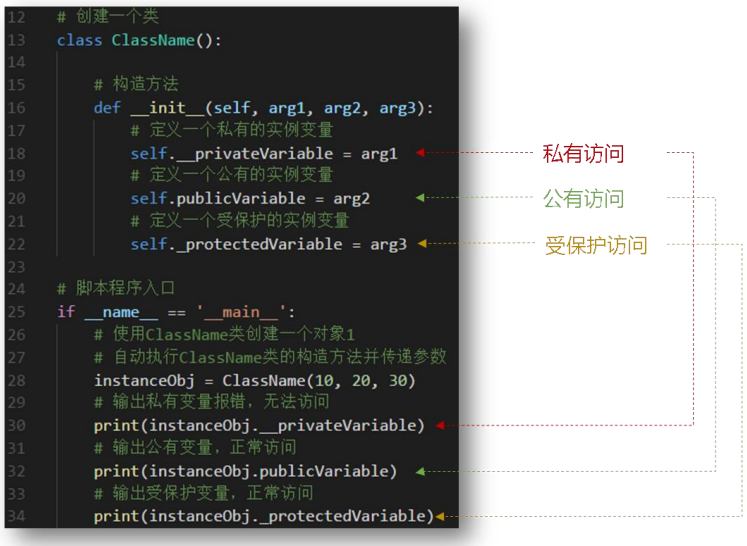
③ 标识符开头单下划线，该变量或方法为 保护权限（即类本身及其子类可以访问调用）

self._name **or** **def** _show(self)

示例 1：不同访问权限的变量

示例：ch10-demo01-permisssion-var.py

定义不同访问权限的实例变量，并在类外部（main 中）进行访问。



示例 2：使用公有的方法访问私有变量

示例：ch10-demo02-permisssion-var2.py

创建公有实例方法，公有实例方法属于类本身，因此其可以访问私有的实例变量。从而达到私有变量对外隐藏，但通过公有方法可以访问的目的。

```
# 定义一个公有的实例方法
def showVariables(self):
    print('调用showVariables()输出：')
    print('>> {0}'.format(self.__privateVariable))
    print('>> {0}'.format(self.publicVariable))
    print('>> {0}'.format(self._protectedVariable))
    pass

# 定义对象输出方法
def __str__(self):
    print('调用__str__输出：')
    print('>> {0}'.format(self.__privateVariable))
    print('>> {0}'.format(self.publicVariable))
    print('>> {0}'.format(self._protectedVariable))
    return ''
```

9.1.4 访问修饰符在方法中的使用

- 同样的道理，访问修饰符下划线，也可以在类的方法中使用；
- 这样，我们的类中方法也进行了调用管控。

示例：ch10-demo03-permisssion-method.py

```
# 创建一个类
class ClassName():

    # 构造方法
    def __init__(self, arg1, arg2, arg3):
        # 定义一个私有的实例变量
        self.__privateVariable = arg1
        # 定义一个公有的实例变量
        self.publicVariable = arg2
        # 定义一个受保护的实例变量
        self._protectedVariable = arg3
        # 在构造方法中调用私有方法
        self.__privateMethod()

    # 定义一个私有方法
    def __privateMethod(self):
        print('>> 我是私有实例方法，仅在类内部可以使用……')
        pass
```

私有实例方法可以在类内部调用

私有实例方法的定义

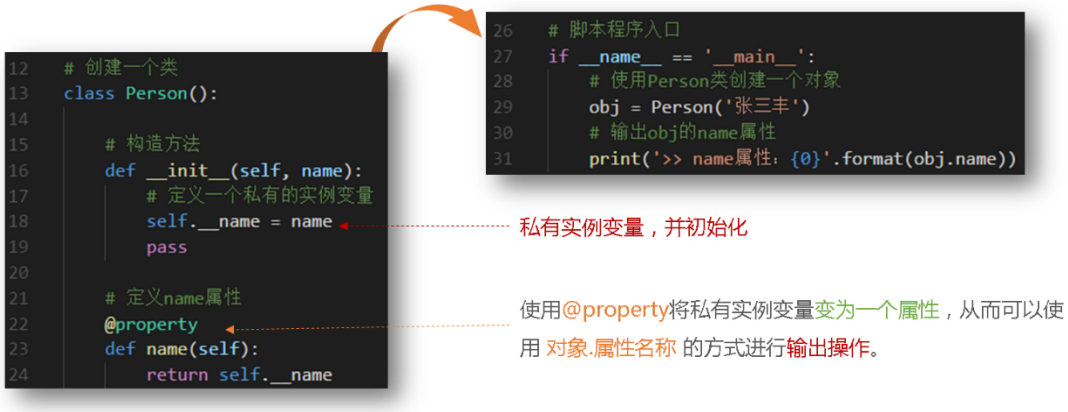
```
# 脚本程序入口
if __name__ == '__main__':
    # 使用ClassName类创建一个对象1
    # 自动执行ClassName类的构造方法并传递参数
    instanceObj = ClassName(10, 20, 30)
    # 实例对象访问私有方法
    instanceObj.__privateMethod()
```

报错，外部不能调用私有

9.1.5 @property 属性装饰器

- 利用面向对象的编程思想，我们会将所有的事物都先抽象成一个类，而类中有描述该事物的属性和行为。
- 之前我们称 属性 为 实例变量（其实不是很好理解）。
- Python 提供了装饰器@property，可以将方法定义成属性，后续可以使用 obj.属性名称 的方式输出。

示例：ch10-demo04-property.py



9.1.6 对象属性的setter和getter

- 为了更好地体现封装，在面向对象编程中创建类的规范：
  ① 所有的属性都必须为私有（安全保护，防止类外直接访问操作）；
  ② 使用公有的 setter（写入）和 getter（读取）方法操作（对外暴露操作，可间接访问私有变量）。

示例：ch10-demo05-property.py

```python
# 创建一个类
class Person():

    # 构造方法
    def __init__(self, name):
        # 定义一个私有的实例变量
        self.__name = name
        pass

    # 定义name属性
    @property
    def name(self):
        return self.__name
```

```python
# 脚本程序入口
if __name__ == '__main__':
    # 使用Person类创建一个对象
    obj = Person('张三丰')
    # 输出obj的name属性
    print('>> name属性：{0}'.format(obj.name))
```

示例 3：定义一个操作对象的标准

一个标准的类在创建时，需遵循以下**四必须**标准规范：

① 类必须定义构造方法 __init__

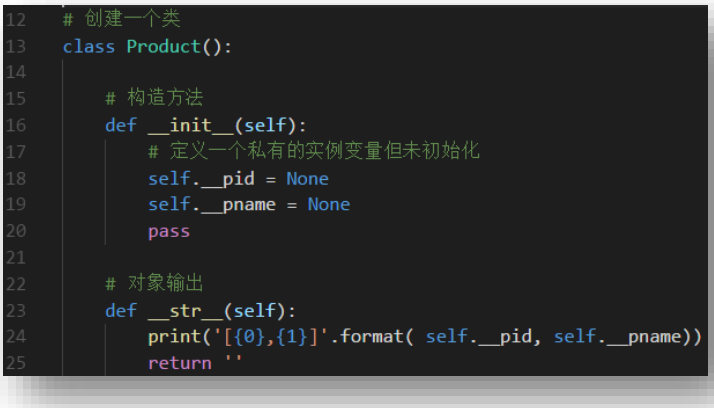
② 类必须定义对象输出 __str__

③ 类属性必须为私有

④ 类必须设置公有属性访问函数

示例：ch10-demo06-example-pojo.py

步骤 1：定义类的构造方法、私有属性以及对象输出方法



```python
# 创建一个类
class Product():

    # 构造方法
    def __init__(self):
        # 定义一个私有的实例变量但未初始化
        self.__pid = None
        self.__pname = None
        pass

    # 对象输出
    def __str__(self):
        print('[{0},{1}]'.format( self.__pid, self.__pname))
        return ''
```

步骤 2：定义属性 pid

```
27      # 定义pid属性
28      @property
29      def pid(self):
30          return self.__pid
31
32      # 定义设置pid的实例方法
33      @pid.setter
34      def pid(self, value):
35          self.__pid = value
36          pass
37
38      # 设置获取pid的实例方法
39      @pid.getter
40      def pid(self):
41          return self.__pid
```

步骤 3：定义属性 pname

```
43      # 定义pname属性
44      @property
45      def pname(self):
46          return self.__pname
47
48      # 定义设置pid的实例方法
49      @pname.setter
50      def pname(self, value):
51          self.__pname = value
52          pass
53
54      # 设置获取pid的实例方法
55      @pname.getter
56      def pname(self):
57          return self.__pname
```

在 main 入口中，创建对象并调用测试

```python
# 脚本程序入口
if __name__ == '__main__':
    # 使用Product类创建一个对象
    obj = Product()
    # 设置Product的pid属性
    obj.pid = 1
    # 设置Product的pname属性
    obj.pname = '测试商品'
    # 输出对象
    print(obj)
    # 输出
    print('pid: ' , obj.pid)
    print('pname: ' + obj.pname)
```

运行结果：

```
[1,测试商品]

pid:  1
pname: 测试商品
```

扩展：快速生成 10 个商品并输出

我们自定义一个函数，使用 for 循环 + yield 生成器，快速完成 10 个商品对象的创建。

```python
#-------- 扩展 --------
# 使用for循环生成10个商品并输出
# 自定义函数实现生成10个商品
def generatorProduct():
    for i in range(10):
        # 创建商品对象
        obj = Product()
        # 商品属性初始化
        obj.pid = i+1
        obj.pname = '测试商品'
        # 使用生成器将创建好的商品对象添加到临时的内存中
        yield obj
```

For循环输出：

```python
# 输出列表中的数据
for product in generatorProduct():
    print(product)
    pass
```

yeild 关键字代表返回一个连续的存储空间地址

9.2 OOP 三大特征之二：继承

继承的实现 / 深度优先和广度优先 / 经典类和新式类

9.2.1 继承 extends

- **继承**，面向对象中的继承和现实生活中的继承相同，即：子可以继承父的内容。

例如：

- 猫可以：喵喵叫、吃、喝、拉、撒
- 狗可以：汪汪叫、吃、喝、拉、撒

我们发现无论是猫还是狗，都有**共同的行为**，同时也有自己个性的行为叫。

9.2.2 继承的语法

- **父类-子类**的纵向编程模式。

```
# 创建父类
class MainClass():
    .............
    pass
```

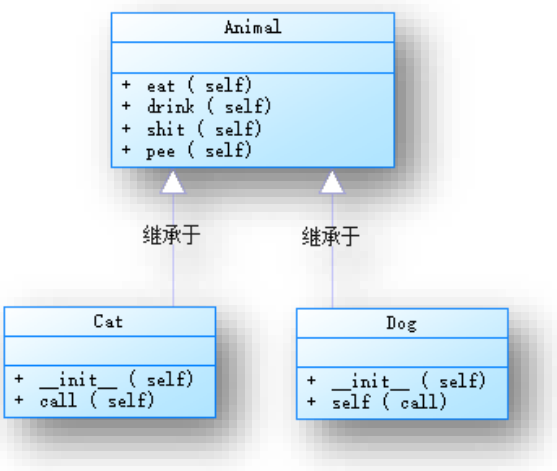
继承表现形式

```
# 创建子类继承父类
class SubClass(MainClass):
    .............
    pass
```

注意，在继承关系下，子类将会拥有父类全部的方法（即，父类方法可在子类对象中访问调用）

示例 (ch10-demo07-extends.py ：

- **UML**，类图设计，Cat 类和 Dog 类继承于 Animal 动物类。



说明：

通过类图我们可以看出：

- Animal 类有 eat（吃）、drink（喝）、shit（拉）、pee（撒）四个公共方法。
- Cat 和 Dog 类在继承了父类 Animal 的同时，又有各自自己的 Call 方法。
- 因此，我们可以得出一个结论：一般共有的方法可以抽象出来放在父类中，子类在继承父类的同时凸显出自己个性的方法。

示例 4：（ch10-demo07-extends.py）

- 创建 Animal 父类，与其子类：Cat

```python
# 定义创建父类Animal
class Animal():
    # 定义四个通用方法
    def eat(self):
        print('%s 吃' % self.name)
        pass
    def drink(self):
        print('%s 喝' % self.name)
        pass
```

```python
# 定义子类Cat继承父类Animal
class Cat(Animal):
    # 定义构造方法
    def __init__(self, name):
        self.name = name
        pass
    # 定义叫call
    def call(self):
        print('%s 妙妙' % self.name)
        pass
```

9.2.3 多重继承

- **Q：那么问题又来了，多继承呢？**

① 是否可以继承多个类？

② 如果继承的多个类每个类中都定了相同的函数，那么那一个会被使用呢？

答：

① Python 的类**可以继承多个类**，不像很多高级编程语言 Java 和 C#那样只能继承一个类（单根语系）。

② **特别注意：**在 Python2.x 版本中，类如果继承了多个类，那么其对象调用寻找方法的方式有两种，分别是：**深度优先** 和 **广度优先**

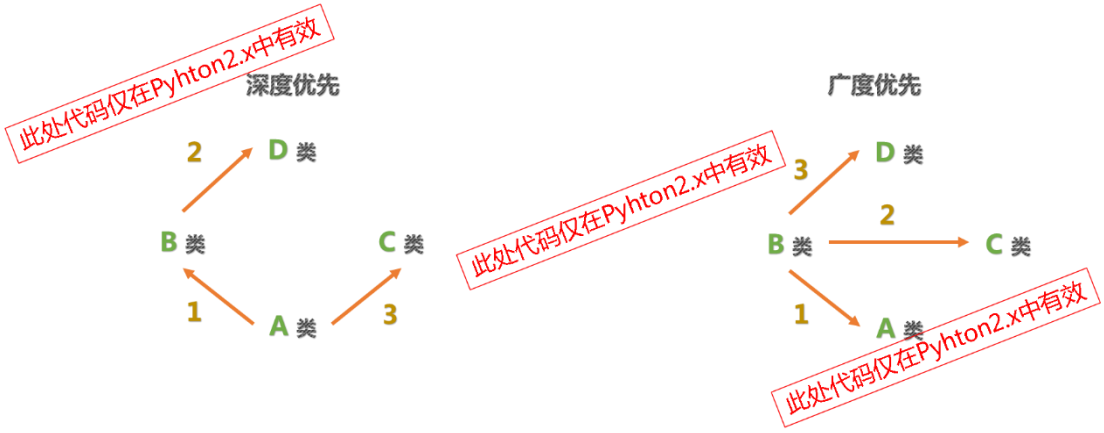
示例 5：（ch10-demo08-extends-methods.py）

- 创建 ClassA 、ClassB 和 ClassC 三个类；
- ClassC 多重继承 ClassA 和 ClassB；
- ClassA 中有三种访问受限的方法。

- 测试 1： 同名方法检索路径
- 测试 2：私有是否可以被继承？

9.2.4 深度优先和广度优先

Python3 中默认均为新式类，以下概念仅在 Python2.7 中有效

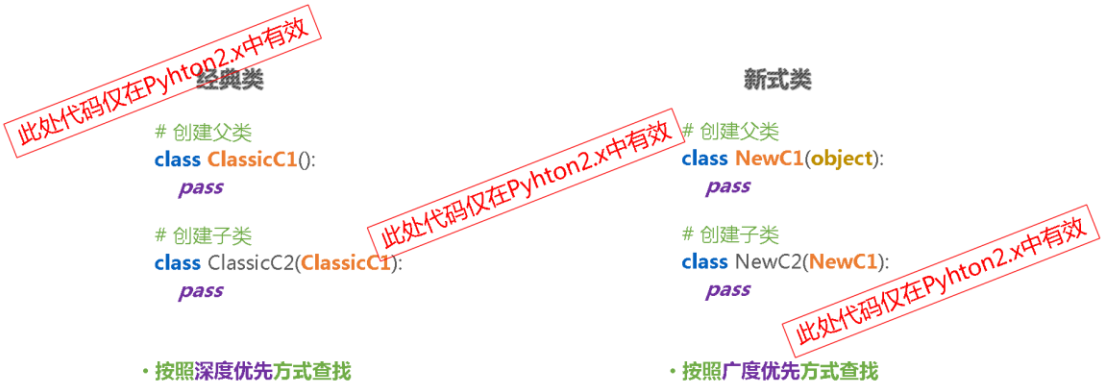


- 当类是经典类时，多继承情况下，会按照深度优先方式查找
- 当类是新式类时，多继承情况下，会按照广度优先方式查找

9.2.5 经典类 和 新式类

Python3 中默认均为新式类，以下概念仅在 Python2.7 中有效

- 经典类和新式类，从字面上可以看出一个老一个新，新的必然包含了跟多的功能，也是之后推荐的写法。
- 从写法上区分的话，如果 **当前类或者父类继承了 object 类**，那么该类便是新式类，否则便是经典类。



9.2.6 继承下的super（）概述

- 在 Python 面向对象的继承特征下，若子类需调用父类的方法，则需要使用 super（）来实现。
- **示例：**

```python
# 定义类ClassA
class ClassA():

    # 定义一个公有方法
    def publicMethod(self):
        print('>> {0}'.format('ClassA的public实例方法'))
        pass

# 定义类ClassB
class ClassB(ClassA):

    # 定义一个公有方法
    def publicMethod(self):
        print('>> {0}'.format('ClassB的public实例方法'))
        super().publicMethod()
        pass
```

```python
# 脚本程序入口
if __name__ == '__main__':
    # 创建classB对象
    b = ClassB()
    # 调用
    b.publicMethod()
```

9.2.7 当super遇到__init__

- 我们以一个房屋作为示例。
- **示例：**

```python
# 定义一个房间类
class Room:
    def __init__(self,area=120, usedfor='sleep'):
        self.area = area
        self.usedfor = usedfor

    def display(self):
        print("this is my house")

# 定义一个婴儿房子类
class BabyRoom(Room):
    def __init__(self,area=40, usedfor="son", wallcolor='green'):
        super().__init__(area,usedfor)
        self.wallcolr = wallcolor

    def display(self):
        super().display()
        print("babyroom area:%s wallcollor:%s"%(self.area,self.wallcolr))
```

```python
# 脚本程序入口
if __name__ == '__main__':
    # 创建classB对象
    babyroom = BabyRoom()
    babyroom.display()
```

9.3 OOP 三大特征之三：多态

继承的实现 / 深度优先和广度优先 / 经典类和新式类

9.3.1 多态的定义

- Python **不支持多态**并且也用不到多态，多态的概念是应用于 Java 和 C#这一类强类型语言中，而 Python 崇尚“**鸭子类型**”。
- **Q：什么是“鸭子类型”？**
- 鸭子类型（英语：duck typing）是动态类型的一种风格。在这种风格中，一个对象有

效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由 James Whitcomb Riley 提出的鸭子测试，“鸭子测试”可以这样表述：

9.3.2 “鸭子类型”的应用

示例 *（demo08.py）*：

```
class F1:
    pass
class S1(F1):
    def show(self):
        print 'S1.show()'
class S2(F1):
    def show(self):
        print 'S2.show()'
```

```
def Func(obj):
    print obj.show()
# 创建S1类对象
s1_obj = S1()
Func(s1_obj)
# 创建S2类对象
s2_obj = S2()
Func(s2_obj)
```

- 为了让 Func 函数既可以执行 S1 对象的 show 方法，又可以执行 S2 对象的 show 方法，所以，定义了一个 S1 和 S2 类的父类
- 而实际传入的参数是：S1 对象和 S2 对象

9.4 本章总结

通过本章节的学习，我们掌握了面向对象思想的三大特征以及 Python 编程语言如何快速实现方法。面向对象的概念是编程语言中的核心概念，本章节通过大量的案例对 Python 编程语言的在面向对象领域的基础应用进行了全面的介绍。

第10章：文件操作基础

知识点

目标 1：文件的基本概念和分类

目标 2：掌握获取文件的相关信息参数

目标 3：掌握文本文件的读写操作

目标 4：掌握二进制文件的读写操作

技能点

实战任务 1：获取文件的相关信息

实战任务 2：文本文件的读写操作

实战任务 3：二进制图片文件读写操作

10.1 文件操作基本原理

10.1.1 文件操作模块库OS

- Python os 模块提供了一个统一的操作系统接口函数（特别对文件）。
- os 模块能在不同操作系统平台（如 nt 或 posix）中的特定函数间自动切换,从而能实现跨平台操作。

10.1.2 文件的定义

- 文件可以认为是相关记录或存放在一起的数据的集合。

- 文件在计算机中可以分为文本文件和二进制文件两类：
  ① 文本文件：在不同操作系统下，可以用文本编辑器进行读写操作的文件。
  ② 二进制文件：那么其他的文件就属于二进制文件。而二进制文件相比与文本文

件的优势在于二进制文件的处理效率更高一些。

10.1.3 文件路径

- 在程序编程中，我们会通过文件的路径来访问（写入操作/读取操作）指定的文件。
- 访问文件路径分为两类方式：

① **绝对路径**

绝对路径是由 驱动器字母+文件所在的完整路径+文件名称 组成的。

- 如果是 Windows 系统，那么某一个文件的绝对路径例如：d:\dev\python\ch01-demo01.py
- 在 Unix 平台上，文件的绝对路径例如：/home/python/ch01-dmeo01.py

③ **相对路径**

是相对与文件当前的工作路径而言，例如： ./ch01-demo01.py

10.2 os 模块介绍

- os 模块 是 Python 编程语言获取文件相关参数（如：文件路径、文件名称、文件大小、修改时间……等）的重要操作模块。**（特别说明：os 模块不参与文件的读写操作）**

- **os 模块** 是 Python 语言的内置模块，但不属于脚本默认导入模块，因此在使用的时候需要使用 import 关键字预先导入后方可使用。

```
>>> import os
```

10.2.1 查看文件路径及文件名称

- 使用 os 模块查看当前脚本文件的目录、文件名称、文件大小等相关参数信息。
- **示例：**ch01-demo01-filepath.py
- 查看当前文件的绝对路径地址
- 函数：os.**path**.realpath(*文件对象*)，该函数返回指定文件对象的绝对访问路径地址。

```
15  # 输出当前脚本文件的绝对路径
16  absolutePath = os.path.realpath(__file__)
17  print('当前文件的路径: {0}'.format(absolutePath))
```

__file__ 为Python语言内置的关键对象，专门获取当前文件对象的引用。

① 查看当前文件的所在目录绝对路径地址

函数：os.**path**.dirname(*文件绝对路径*)，该函数返回指定文件对象所在目录的绝对路径地址。

```
19  # 从当前文件路径中获取文件目录
20  absoluteDir = os.path.dirname(absolutePath)
21  print('当前文件的目录: {0}'.format(absoluteDir))
```

- **示例：**ch01-demo01-filepath.py

③ 查看当前文件的名称

函数：os.**path**.basename(*文件绝对路径*)，该函数返回指定文件对象的名称。

```
23  # 从当前文件路径中获取文件名称
24  fileName = os.path.basename(absolutePath)
25  print('当前文件的名称: {0}'.format(fileName))
```

④ 查看当前工程文件夹的绝对路径地址

函数：os.getcwd( )，该函数返回当前工程文件夹的绝对路径地址。

```
28  # 获取当前工程的所在目录
29  projectDir = os.getcwd()
30  print('当前工程文件夹: {0}'.format(projectDir))
```

- **示例：**ch01-demo01-filepath.py

③ 查看当前文件的大小

函数：os.**path**.getsize(*文件绝对路径*)，该函数返回指定文件对象的大小（单位为字节）。

```
32    # 获取指定路径文件的大小
33    fileSize = os.path.getsize(absolutePath)
34    print('当前文件的大小: {0} 字节'.format(fileSize))
```

10.2.2 获取文件的创建、修改以及最后访问时间

- **示例：**ch01-demo02-time.py

① 查看当前文件创建时间

函数：os.**path**.getctime(*文件路径*)，返回一个时间戳

```
45    # 获取文件的创建时间
46    createTime = os.path.getctime(absolutePath)
47    print('当前文件创建时间: {0}'.format(createTime))
```

运行结果：

时间戳（timestamp），一个能表示一份数据在某个特定时间之前已经存在的、 完整的、 可验证的数据,通常是一个字符序列，唯一地标识某一刻的时间。使用数字签名技术产生的数据，签名的对象包括了原始文件信息、签名参数、签名时间等信息。广泛的运用在知识产权保护、合同签字、 金融帐务、 电子报价投标、 股票交易等方面。

扩展：时间戳格式化输出

我们使用 time 模块，获取时间戳数据的 timeStruct 时间结构（年、月、日、时、分、秒 等信息）。

自定义一个函数，实现将时间戳数据转换成格式化日期数据：

```python
# 导入time模块
import time
# 定义一个函数时间戳格式化函数
def TimeStampToTime(timestamp):
    # 获取时间属性(年月日时分秒等结构)
    timeStruct = time.localtime(timestamp)
    # 格式化输出返回
    return time.strftime("%Y-%m-%d %H:%M:%S", timeStruct)
```

修改源代码：

```python
# 获取文件的创建时间
createTime = os.path.getctime(absolutePath)
print('当前文件创建时间: {0}'.format(TimeStampToTime(createTime)))
```

- **示例：**ch01-demo02-time.py

② 查看当前文件的访问时间

函数：os.**path**.getatime(*文件路径*)，返回一个时间戳，需格式化输出

```python
# 获取文件的访问时间
accessTime = os.path.getatime(absolutePath)
print('当前文件访问时间: {0}'.format(TimeStampToTime(accessTime)))
```

③ 查看当前文件的修改时间

函数：os.**path**.getmtime(*文件路径*)，返回一个时间戳，需格式化输出

```python
# 获取文件的修改时间
modifyTime = os.path.getmtime(absolutePath)
print('当前文件修改时间: {0}'.format(TimeStampToTime(modifyTime)))
```

10.2.3 判断文件夹或文件

当我们访问到一个文件时，需要知道该文件是一个 文件夹？ 还是一个 文件？

我们需要使用 os.**path**.isdir(*文件对象*) 函数判断，该函数返回 bool 类型

- **示例：**ch01-demo03-isdir.py

判断当前文件路径是否为文件夹？

函数：os.listdir( )，返回一个列表对象，若参数为空，则返回当前工程文件夹中的所有文件名称。

10.2.4 获取文件夹的信息

- **示例：**ch01-demo04-dirinfo.py

① 查看当前文件夹的中的文件名称（默认获取文件夹或文件名称，不会继续查看文件夹中的子文件内容）

函数：os.listdir( )，返回一个列表对象，若参数为空，则返回当前工程文件夹中的所有文件名称。

10.2.5 获取文件夹的信息 os.walk()函数

我们习惯使用递归方式遍历文件夹，在 os 模块中的 walk( )函数也能完成类似的功能，实现起来更加轻松。

函数原型

os.**walk**(*文件地址*) ：返回 文件路径地址（字符串）、文件夹集合（列表）和 文件夹中的文件（列表）

常规使用

我们经常使用 **for 循环** + **os.walk()函数** 组合的形式，实现对指定文件夹的深度遍历。

- **示例：**ch01-demo04-dirinfo.py

```
# 使用os.walk()函数
for root, dirs, files in os.walk('D:\dev\python\workspace\M1_Course02_Demos'):
    print('|-文件夹路径: %s' % root)   # 获取文件夹中的所有目录及子目录名称
    print('| |-路径下的文件夹: {0}'.format(dirs))  # 获取当前文件夹中的子文件夹名称
    print('| |  |-文件夹中的文件: {0}'.format(files))  # 获取当前子文件夹中的文件名称
    pass
```

10.2.6 实战任务：创建获取文件相关信息

需求：客户端接收用户输入的文件路径地址，根据地址显示出该文件的基本信息。

运行效果：

- 测试输入文件夹地址

```
对象名称: CH01_Demos
所在目录: D:\dev\python\workspace\M1_Course02_Demos
文件类型: 文件夹
对象的大小: 7410 字节
对象创建时间: 2018-04-22 10:46:26
对象访问时间: 2018-04-22 23:38:12
对象修改时间: 2018-04-22 23:38:12
```

- 测试输入文件地址

```
对象名称: ch01-demo05-example-fileinfo.py
所在目录: D:\dev\python\workspace\M1_Course02_Demos\CH01_Demos
文件类型: 文件
对象的大小: 3075 字节
对象创建时间: 2018-04-22 20:37:05
对象访问时间: 2018-04-22 23:39:46
对象修改时间: 2018-04-22 23:39:46
```

步骤 1：创建功能函数

1-1：导入所需模块

```
# 导入模块
import os
import time
from os.path import join, getsize
```

1-2：创建时间准换工具函数

```
'''
    @name: TimeStampToTime
    @args: timestamp
    @return: str
    @date: 2018-04-23
'''
def TimeStampToTime(timestamp):
    '定义一个函数时间戳格式化函数'
    # 获取时间属性(年月日时分秒等结构)
    timeStruct = time.localtime(timestamp)
    # 格式化输出返回
    return time.strftime("%Y-%m-%d %H:%M:%S", timeStruct)
```

```
'''
    @name: calcFileSize
    @args: str
    @return: int
    @date: 2018-04-23
'''
def calcFileSize(absolutePath):
    ' 计算文件的大小 '
    print('开始计算文件大小……')
    # 获取文件的大小
    size = os.path.getsize(absolutePath)
    # 返回
    return size
```

步骤 2：创建计算文件和文件夹大小的函数

2-1：创建函数实现计算文件大小

```
'''
    @name: calcFileSize
    @args: str
    @return: int
    @date: 2018-04-23
'''
def calcFileSize(absolutePath):
    ' 计算文件的大小 '
    print('开始计算文件大小……')
    # 获取文件的大小
    size = os.path.getsize(absolutePath)
    # 返回
    return size
```

2-2：创建文件夹大写的计算函数

```python
'''
    @name: calcDirSize
    @args: str
    @return: int
    @date: 2018-04-23
'''
def calcDirSize(absolutePath):
    ' 输出文件夹的大小及及操作日期数据 '
    # 获取文件的大小
    fileSize = os.path.getsize(absolutePath)
    # 使用 os.walk() 函数深度遍历文件夹中的文件
    size = 0 # 累加子文件大小
    for root, dirs, files in os.walk(absolutePath):
        # 输出遍历的文件
        print('|-文件夹路径: %s' % root)   # 获取文件夹中的所有目录及子目录名称
        print('| |-路径下的文件夹: {0}'.format(dirs))  # 获取当前文件夹中的子文件夹名称
        print('| |  |-文件夹中的文件: {0}'.format(files))  # 获取当前子文件夹中的文件名称
        # 使用 sum( 数字序列 )函数
        size += sum([getsize(os.path.join(root, name)) for name in files])
    # 返回
    return size
```

步骤 3：创建显示文件信息的函数

```python
'''
    @name: getFileInfo
    @args: str
    @return: none
    @date: 2018-04-23
'''
def fileDetials(filePath):
    ' 输出文件基本信息 '
    os.system('cls') # 清屏操作
    # 获取文件的绝对路径
    absolutePath = os.path.realpath(filePath)
    # 判断文件路径的文件类型
    fileType = '文件夹' if os.path.isdir(absolutePath) else '文件'
    # 计算文件对象大小
    # 异常处理
    try:
        # 根据文件类型输出不同结果
        size = calcDirSize(absolutePath) \
            if fileType == '文件夹' \
            else calcFileSize(absolutePath)
    except FileNotFoundError:
        print('您输入的文件或文件夹路径有无，请核实后再输入')
        os._exit(0) # 退出系统
```

```
    # 获取文件的名称
    fileName = os.path.basename(absolutePath)
    # 获取文件的所在目录
    dirPath = os.path.dirname(absolutePath)
    # 输出
    print('\n~~~~~~~~~~文件对象信息~~~~~~~~~~')
    print('对象名称: {0}'.format(fileName))
    print('所在目录: {0}'.format(dirPath))
    print('文件类型: {0}'.format(fileType))
    print('对象的大小: {0} 字节'.format(size))

    # 获取文件的创建时间
    createTime = os.path.getctime(absolutePath)
    print('对象创建时间: {0}'.format(TimeStampToTime(createTime)))
    # 获取文件的访问时间
    accessTime = os.path.getatime(absolutePath)
    print('对象访问时间: {0}'.format(TimeStampToTime(accessTime)))
    # 获取文件的修改时间
    modifyTime = os.path.getmtime(absolutePath)
    print('对象修改时间: {0}'.format(TimeStampToTime(modifyTime)))
    pass
```

步骤 4：创建显示文件信息的函数

```
'''
    @name: getFileInfo
    @args: str
    @return: none
    @date: 2018-04-23
'''
def fileDetials(filePath):
    ' 输出文件基本信息 '
    os.system('cls') # 清屏操作
    # 获取文件的绝对路径
    absolutePath = os.path.realpath(filePath)
    # 判断文件路径的文件类型
    fileType = '文件夹' if os.path.isdir(absolutePath) else '文件'
    # 计算文件对象大小
    # 异常处理
    try:
        # 根据文件类型输出不同结果
        size = calcDirSize(absolutePath) \
            if fileType == '文件夹' \
            else calcFileSize(absolutePath)
    except FileNotFoundError:
        print('您输入的文件或文件夹路径有无，请核实后再输入')
        os._exit(0) # 退出系统
```

```
    # 获取文件的名称
    fileName = os.path.basename(absolutePath)
    # 获取文件的所在目录
    dirPath = os.path.dirname(absolutePath)
    # 输出
    print('\n~~~~~~~~~~文件对象信息~~~~~~~~~~')
    print('对象名称：{0}'.format(fileName))
    print('所在目录：{0}'.format(dirPath))
    print('文件类型：{0}'.format(fileType))
    print('对象的大小：{0} 字节'.format(size))

    # 获取文件的创建时间
    createTime = os.path.getctime(absolutePath)
    print('对象创建时间：{0}'.format(TimeStampToTime(createTime)))
    # 获取文件的访问时间
    accessTime = os.path.getatime(absolutePath)
    print('对象访问时间：{0}'.format(TimeStampToTime(accessTime)))
    # 获取文件的修改时间
    modifyTime = os.path.getmtime(absolutePath)
    print('对象修改时间：{0}'.format(TimeStampToTime(modifyTime)))
    pass
```

10.2.7 文件夹的创建和删除

- 文件夹的创建和删除
- **示例：**ch01-demo06-diropt.py

```
# 导入os模块
import os

# 判断文件夹是否存在
if not os.path.exists('createDir'):
    # 创建一个文件夹
    mkdir('createDir')

# 删除文件
os.rmdir('createDir')
```

10.3 文件读写操作

① open 函数

② with 语句

③ 文件写入操作

④ 文件读取操作

⑤ 二进制文件操作

10.3.1 读取文件

- 读取一个文件的思路永远都是相同的：
    - 第 1 步 自然就是打开一个文件。

- 在 python 中我们使用 **open 函数**来打开一个文件。
- 在取得文件关联后，才可以执行文件的写入或读取操作。

- **input = open**（文件路径 ， 读写模式 [, encoding=编码格式]）

10.3.2 open参数说明

- 我们读取文件的路径有两种方式：
  - 绝对路径方式

  input = open（'d:/dev/python/workspace/test.txt'， **'r'**）
  - 相对路径方式

  input = open（'test.txt'， **'r'**）

说明：open( ) 在获取文件关联时，若文件不存在则创建文件。

- 同样我们读取文件的模式主要有 5 种：

模式	作用
r	读取模式
w	写入模式
a	追加模式
rb	二进制数据读取模式
wb	二进制数据写入模式

- 还有一种写法是在常规模式后面添加 + 号，如 r+、w+、a+、rb+、wb+ 等
- + 号的作用是赋予模式在原有功能的基础初上补充并完善所有访问操作功能。
  - 如 r+ 代表就是读写文件模式，不仅仅是只读模式。

10.3.3 使用 write( ) 函数实现文件写入

① 向文件写入一段文字

函数：write(*字符串*)

标准步骤：

- 步骤 1：获取程序与文件的关联
- 步骤 2：写入数据

- 步骤 3：关闭文件对象

示例：ch01-demo07-filewrite.py

```
12  # 导入os模块
13  import os
14
15  fileDir = os.path.join(os.getcwd(), 'CH01_Demos/testDir')
16  print(fileDir)
17
18  # 程序与文件取得关联
19  input = open(fileDir + os.sep + 'a.txt', 'w', encoding='utf8')
20  # 向文件写入数据
21  input.write('我在学习Python\n')
22  input.write('好好学习，天天向上')
23  # 关闭文件流对象
24  input.close()
25  print('文件写入完毕')
```

② 使用 **with 语句**，很好地处理了上下文环境和异常情况，自动释放对象内存。

方式：with open(*文件路径* , *读写模式*) as 文件对象:

示例：ch01-demo08-with.py

```
18  # 程序与文件取得关联
19  with open(fileDir + os.sep + 'a.txt', 'w', encoding='utf8') as fp:
20      # 向文件写入数据
21      fp.write('我在学习Python\n')
22      fp.write('好好学习，天天向上')
23      print('文件写入完毕')
```

说明：取消了 close()，with 语句会在执行完毕后，自动释放 fp 对象

将 Python 语言的简洁性，体现的淋漓尽致。 尝试追加写入 **w** 的效果……

10.3.4 使用read( ) 函数实现文件读取

- **使用 read ( ) 函数实现向已关联的文件读取数据**
- 从指定的文件中读取数据

函数：read( )

标准步骤：

- 步骤 1：获取程序与文件的关联
- 步骤 2：读取并输出数据
- **示例：**ch01-demo08-fileread.py

```
12  # 导入os模块
13  import os
14
15  fileDir = os.path.join(os.getcwd(), 'CH01_Demos/testDir')
16  print(fileDir)
17
18  # 程序与文件取得关联
19  with open(fileDir + os.sep + 'a.txt', 'r', encoding='utf8') as fp:
20      # 向文件写入数据
21      content = fp.read()
22      # 输出数据
23      print(content)
```

10.3.5 二进制文件的读写操作

- **实现图片文件的复制**

标准步骤：

- 步骤 1：读取文件数据
- 步骤 2：写入文件数据

示例：ch01-demo10-binaryfile.py

10.4 本章总结

通过本章的学习，我们基本掌握了 Python 编程语言如何实现文件的读写操作。通过对标准流程的固化以及对实际操作的应用技巧介绍，为我们后续学习各种常用数据文件的操作打下了坚实的基础。

第11章：常用数据文件操作

知识点

目标 1：Python 序列化操作

目标 2：Python 对 json 文件的各项操作

目标 3：Python 对 csv 数据文件的各项操作

目标 4：掌握 Python 对 excel 数据表格的读写操作

技能点

实战任务 1：Python 读写 json 数据文件

实战任务 2：Python 读写 csv 数据文件

实战任务 3：Python 读写 excel 数据文件

11.1 对象序列化

① 序列化的定义和原理

② 序列化操作的意义

③ 序列化的应用场景

11.1.1 数据对象的序列化

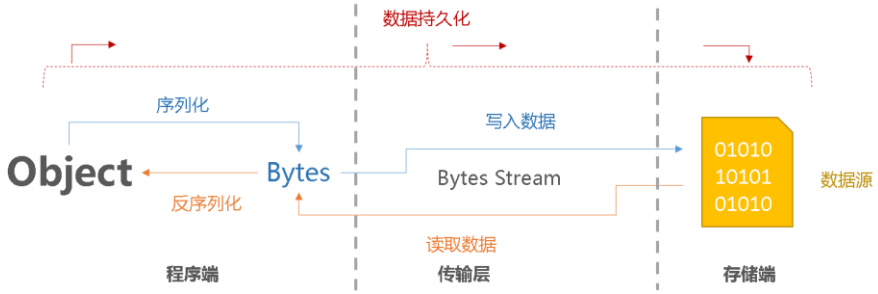
- 什么是序列化？

所谓序列化，指的是将编程语言中的各种对象转换成字节流的过程。同时，也可以逆向操作将字节流还原成一个对象，这个过程称之为反序列化。

11.1.2 序列化操作的意义

- **为什么使用序列化？**

能够让对象永久性保存到诸如文件、数据库等数据源中，我们需要将对象首先变为字节流，这样就可以通过文件读写操作或数据库读写操作完成数据的永久性存储（这个过程也称为 数据持久化操作）。



11.1.3 序列化应用的场景

- **什么时候使用序列化？**

我们经常会将程序的数据存储在如列表、字典等序列对象中临时存储，但是一旦程序结束，这些对象就会从内存中消失，无法永久性的保存下来。

1. 数据写入文本文件
2. 二进制文件读写
3. 数据写入数据库

……

总言之，数据写入任何可以写入的、可保存的介质中存储时……

11.2 Python 序列化操作

① pickle 模块介绍

② pickle 常用函数 API

11.2.1 Pickle 模块介绍

- 在 Python 编程语言中，**pickle 模块** 是实现 Python Object 序列化的重要操作模块。

- **Pickle 模块** 是支持将一个 python 对象转换成字节流，同时还能将对应的字节流逆操作，得到原来的对象。
- **pickle 模块** 的常用函数：
  ① pickle.dump(obj, file [,protocol]) 对象序列化操作函数
  ② pickle.load(file) 将 file 中的对象序列化读取
  ③ pickle.dumps(obj, [, protocol]) 将对象转化成 String 字符串类型
  ④ pickle.loads(string) 从 string 中读取序列化前的 obj 对象

11.2.2 Pickle 模块API函数应用

- **Pickle 模块** 属于 Python 语言的内置模块，需要使用 import pickle 导入后方可使用。
- pickle.dump(obj, file, [,protocol])
- **作用：**将 **obj** 对象进行序列化操作，并写入到 **file** 引用文件对象中。

- **补充：**将要持久化的数据“对象”，保存到“文件”中，使用有 3 种协议，索引 0 为 ASCII，1 为旧式二进制，2 为新式二进制协议（不同之处在于 2 要更高效一些，默认 dump 方法使用 0 做协议）

示例：使用 dump 函数完成字典对象的序列化存储操作 （ch02-demo01-dump-load.py）

```
# 将字典对象写入文件
with open(filePath + os.sep + 'data.txt', 'wb') as fp:
    # 使用pickle中的dump()函数将字典对象序列化并保存到文件
    pickle.dump(dict, fp)
    print('>> 数据写入完毕.')
```

序列化是一个典型的字节数据转化过程，因此文件写入模式应为“二进制”读写模式。

- pickle.load(file)
- **作用：**将 **file** 中的数据读取并反序列化还原成之前的对象。
- **补充：**从“文件”中读取字符串，将他们反序列化转换为 python 的数据对象，可以像操作数据类型的这些方法来操作它们。

示例：使用 load 函数实现读取数据文件并进行反序列化操作（ch02-demo01-dump-load.py）

```
# 读取文件数据
with open(filePath + os.sep + 'data.txt', 'rb') as fp:
    # 使用pickle中的load()函数加载文件数据并反序列化成对象
    obj = pickle.load(fp)
    print('>> obj-> {0}'.format(type(obj)))
    print(obj)
```

- pickle.dumps(obj [, protocol])
- **作用：**将 **obj** 对象转换成一个字符串数据，但并不存入文件中。

- **补充：** protocol 参数如果该项省略，则默认为 0。如果为负值或 HIGHEST_PROTOCOL，则使用最高的协议版本。

示例：使用 dumps 函数将列表对象进行序列化处理（ch02-demo02-dumps-loads.py）

```
# 定义一个列表对象
lstInfo = [1, 2, 3, 4, 'abc', True]

# 使用dumps()函数进行对象序列化操作
data1 = pickle.dumps(lstInfo)
print('lstInfo序列化数据：', end=' ')
print(data1)
```

- pickle.loads(string)
- **作用：**将 **string** 字符串反序列化还原成之前的对象。

示例：使用 loads 函数将字符串反序列化还原成之前的对象（ch02-demo02-dumps-loads.py）

```
# 使用loads()函数进行反序列化操作
data2 = pickle.loads(data1)
print('lstInfo反序列化数据：', end=' ')
print(data2)
```

11.2.3 实战任务：OOP对象序列化数据存储

① OOP 面向对象

② Pickle 模块序列化操作

③ io.BytesIO 内存字节处理（扩展）

业务需求

- 用户控制台输入员工编号 empno，员工姓名 ename 和基本工资 salary 数据。
- 将输入的数据对象写入到数据文件中。
- 查看数据文件中的所有数据。

技术需求

- 创建员工 Employee 类，并设置属性；
- 员工数据对象 写入或读取 至数据文件 employees.txt

步骤 1：创建员工类 Employee

- 创建员工 Employee 类；
- 设置员工编号和员工姓名两个属性；
- 重写__str__方法。

```python
# 员工类
class Employee():

    # 构造方法
    def __init__(self, empno, ename):
        self.__empno = empno # 员工编号
        self.__ename = ename # 员工姓名
        pass

    # 对象输出
    def __str__(self):
        print('empno: {0}'.format(self.__empno))
        print('ename: {0}'.format(self.__ename))
        return ''
```

步骤 2：创建员工类 EmployeeDao

- 实现添加员工的方法 **addEmployee**(self, employee, filePath)；
- 实现查询员工的方法 **findAllEmployee**(self, filePath).

```python
# 员工操作类
class EmployeeDao():

    # 构造方法
    def __init__(self):
        pass

    # 添加员工的实例方法
    def addEmployee(self, employee, filePath):
        # 使用with语句完成二进制文件写入
        with open(filePath, 'wb') as fp:
            # 使用dump()序列化并写入文件
            pickle.dump(employee, fp)
            print('提示: 员工添加成功.')
            pass

    # 读取员工数据并显示
    def findAllEmployee(self, filePath):
        # 使用with语句完成二进制文件读取
        with open(filePath, 'rb') as fp:
            # 使用load()读取数据文件并反序列化成对象
            data = pickle.load(fp)
            # 返回结果
            print(data)
```

步骤 3：脚本入口

```python
# 脚本入口
if __name__ == '__main__':
    # 设置数据文件的绝对路径地址
    filePath = os.path.join(os.getcwd(), 'CH02_Demos/data')

    print('#' * 30)
    print('添加员工信息')
    print('#' * 30)
    # 接收用户输入
    empno = input('empno: ')
    ename = input('ename: ')
    # 创建Employee对象并为属性初始化
    emp = Employee(empno, ename)
    # 创建EmployeeDao操作对象
    empDao = EmployeeDao()
    # 调用empDao对象中的addEmployee()实现对对象持久化存储
    empDao.addEmployee(emp, filePath + os.sep + 'employee.info')
    # 调用empDao对象时中的selectAllEmployee()实现获取对象
    empDao.findAllEmployee(filePath + os.sep + 'employee.info')
    pass
```

补充：StringIO 和 BytesIO

- 很多时候，数据读写不一定是文件，也可以在内存中读写。
- 这时候就需要 StringIO（） 和 BytesIO（）
    - StringIO（）创建内存空间并写入字符串数据 str
    - BytesIO（）创建内存空间并写入字节数据 Bytes

- **io.BytesIO 使用方式：**

步骤 1：导入模块

```
from io import BytesIO
```

步骤 2：创建内存空间

```
f = BytesIO（）
```

步骤 3：写入字节数据操作

```
f.write（字节数据）
```

步骤 4：读取字节数据操作

```
f.getvalue()
```

```
77    # 导入BytesIO
78    from io import BytesIO
79    # 创建内存字节存储空间
80    f = BytesIO()
81    # 将employee对象写入
82    f.write(pickle.dumps(emp))
83    # # 获取序列化后的员工字节数据
84    e = f.getvalue()
85    print(e)
86    # 得到反序列化后的员工对象数据
87    e = pickle.loads(e)
88    print(e)
```

11.3 Json 文件格式存储

① json 模块介绍

② json 常用函数 API

11.3.1 JSON 数据

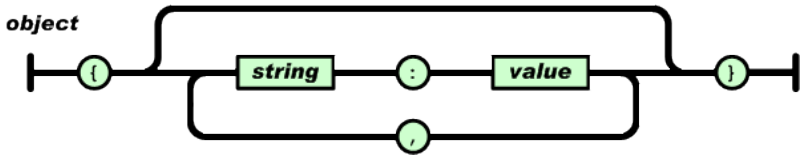
- **JSON** (JavaScript Object Notation) 是一种轻量级的数据交换格式，广泛地应用于数据存储。
- JSON 建构于两种结构：

① “名称/值”对的集合（A collection of name/value pairs)。不同的语言中，它被理解为对象(object)，纪录(record)，结构(struct)，字典(dictionary)，哈希表(hash table)，有键列表（keyed list)，或者关联数组 （associative array)。

② 值的有序列表(An ordered list of values)。在大部分语言中，它被理解为数组(array)。

3.2 JSON 数据格式 1/2

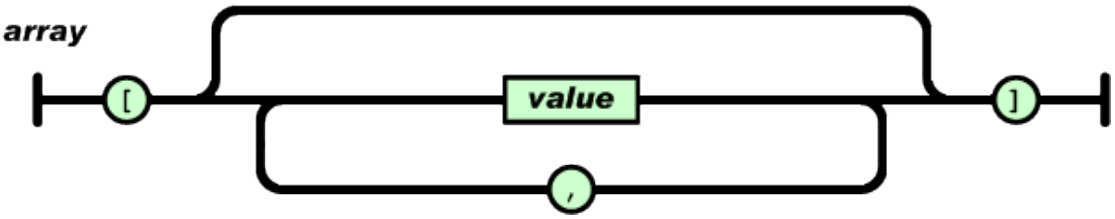
- 对象是一个无序的“‘名称/值’对”集合。一个对象以**“{”**（左括号）开始，**“}”**（右括号）结束。每个“名称”后跟一个**“:”**（冒号）；“‘名称/值’ 对”之间使用**“,”**（逗号）分隔。



- **举例：**
- 员工对象：Employee 中，员工编号为 1001，员工姓名为 Jack
- JSON 描述格式：{ ‘empno’: ‘1001’, ‘ename’: ‘Jack’ }

11.3.2 JSON 数据格式

- 数组是值（value）的有序集合。一个数组以**“[”**（左中括号）开始，**“]”**（右中括号）结束。值之间使用**“,”**（逗号）分隔。



- **举例：**

员工序列对象中包含多个 Employee 对象

JSON 描述格式：[{ ‘empno’: ‘1001’, ‘ename’: ‘Jack’ },
{ ‘empno’: ‘1002’, ‘ename’: ‘Nacy’ } ,
{ ‘empno’: ‘1003’, ‘ename’: ‘Ford’ }]

11.3.3 json 模块

- Python 语言使用内置的 **json 模块** 完成对 json 文件的解析操作。需要使用 import json 预先导入。

- 它包含两个常用函数：

① json.dumps( ) 完成数据对象的 json 格式序列化操作，返回一个 json 串对象

② json.loads（）完成对 json 数据的反序列化，返回一个原始对象

11.3.4 Python 中的 json 对象转换

- Python 语言使用内置的 **json 模块** 完成对 json 文件的解析操作。需要使用 import json 预先导入。

表1：Python序列化编码转换为JSON类型对应表

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

表2：JSON反序列化编码转换成Python类型对应表

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

11.3.4 JSON 模块API函数应用

- json.dumps(obj)
- **作用：**将 obj 对象进行序列化编码为 JSON 格式。
- **示例：**使用 dumps 函数完成字典对象的序列化存储操作 （ch02-demo04-json-dumps-loads.py）

```
# 导入json模块
import json

# 定义一个字典类型对象
data = {'pid':'p001', 'pname':'测试商品名称', 'price':299}
# 使用json.dumps()函数完成对象序列化转换
strJson = json.dumps(data, ensure_ascii=False)

# 测试输出
print('data原始数据: {0}'.format(data))
print('json转换数据: {0}'.format(strJson))
```

ensure_ascii 属性设置False，防止中文数据乱码

- json.loads(json_str)
- **作用：**将 JSON 字符串 编码转换成 原始对象。
- **示例：**使用 loads 函数完成 json 字符串的对象类型转换（ch02-demo04-json-dumps-loads.py）

```
# 使用json.loads()函数完成json->对象的转换
obj = json.loads(strJson)
print('obj -> {0}'.format(type(obj)))
print('pid: {0}'.format(data['pid']))
print('pname: {0}'.format(data['pname']))
print('price: {0}'.format(data['price']))
```

- json.dump(obj , file) 和 json.load(file)
- **作用：**将序列编码后的 json 字符串写入文件 和 从文件读取字符串反序列化编码成对象。
- **示例：**使用 dump 函数（ch02-demo05-json-dump-load.py）

```
# 定义一个字典类型对象
data = {'pid':'p001', 'pname':'测试商品名称', 'price':299}

# 写入json文件
with open(filePath + os.sep + 'data.json' , 'w', encoding='utf8') as fp:
    # dump() 函数写入json数据
    json.dump(data, fp)
    print('json文件数据写入完毕.')
    pass
```

示例：使用 load 函数（ch02-demo05-json-dump-load.py）

```
# 读取json文件
with open(filePath + os.sep + 'data.json' , 'r', encoding='utf8') as fp:
    # dump() 函数写入json数据
    data = json.load(fp)
    # 输出
    print('dataType-> {0}'.format(type(data)))
    print('>> {0}'.format(data))
    pass
```

11.4 CSV 文件格式存储

① CSV 数据格式介绍

② CSV 模块介绍

③ CSV 常用函数 API

11.4.1 CSV 数据

- 逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。
- CSV 文件由任意数目的记录组成，记录间以某种换行符分隔；
- 每条记录由字段组成,字段间的分隔符是其它字符或字符串,最常见的是逗号或制表符。
- CSV 数据文件经常用于数据统计和数据分析过程中的数据存储格式

11.4.2 csv模块介绍

- **csv 模块** 是 Python 的内置模块，需要引用后方可使用 import csv。
- **csv 模块** 提供了各种函数帮助我们快速完成对 csv 数据的读写操作。
- **csv 模块** 的常用函数 API：
    - **reader(……)** 读取 csv 数据
    - **writer(……)** 写入 csv 数据
    - DictReader(……) 读取 csv 数据并可转换成字典数据类型
    - DictWriter(……) 读取字典数据并写入 csv 文件

11.4.3 csv文件读写操作

- **csv.reader(** csv_file **)**
- **作用：**获取一个可迭代的数据类型 _csv.reader ， 我们可以使用 for 循环遍历输出该类型对象。

示例：使用 reader 函数读取 csv 文件并输出（ch02-demo06-csv-reader-writer.py

```
25  # 使用with结构改造
26  with open('CH02_Demos/data/data.csv', 'r', encoding='utf8') as csvfile:
27      # 使用reader()函数读取csv数据
28      reader = csv.reader(csvfile)
29      print('>> reader Type -> {0}'.format(reader))
30      # 使用for循环迭代reader对象
31      for line in reader:
32          print(line)
```

- **csv.writer(** csv_file **)**

- **作用：**获取一个 csv 写入对象 _csv.writer。
    - 通过该对象的 writerow（数据对象）完成单行数据写入；
    - 通过该对象的 writerows（数据对象）完成多行数据写入。

示例：使用 write 函数写入 csv 文件（ch02-demo06-csv-reader-writer.py）

```
# 写入csv文件
# 使用write()函数写入数据
colums = ['姓名', '年龄', '电话']
# 写入数据
data = [('测试人员1', 18, '1388888888'),
        ('测试人员2', 22, '1366666666'),
        ('测试人员3', 20, '1399999999')]
```

- **csv.writer(** csv_file **)**
- **作用：**获取一个 csv 写入对象 _csv.writer。
    - 通过该对象的 writerow（数据对象）完成单行数据写入；
    - 通过该对象的 writerows（数据对象）完成多行数据写入。

示例：使用 write 函数写入 csv 文件（ch02-demo06-csv-reader-writer.py）

```
# 使用with语句打开文件关联
with open('CH02_Demos/data/data2.csv', 'w', encoding='gb18030', newline='') as csvfile
    # 获取writer写入对象
    writer = csv.writer(csvfile)
    # 写入数据标题行（单行数据）
    writer.writerow(colums)
    print('数据标题项写入完毕.')
    # 使用writer对象的writerows()一次性写入多行数据
    writer.writerows(data)
    print('多行数据写入完毕.')
```

11.4.4 DictReader()/DictWriter()

- **csv.DictReader(** csv_file **)**
- **作用：**获取一个 csv 读取对象 _csv.dictreader。同时其获取后的数据可以使用 dict(obj) 直接转换成字典数据类型。

示例：使用 DictReader 函数读取 csv 数据并转化成字典输出(ch02-demo07-csvdictreader.py)

```
17  # 读取csv数据并转成字典数据类型
18  with open('CH02_Demos/data/data2.csv', 'r') as csvfile:
19      # 使用DictReader()函数读取csv数据
20      reader = csv.DictReader(csvfile)
21      print('>> reader type -> {0}'.format(reader))
22      # 使用for循环迭代reader对象
23      for line in reader:
24          print(dict(line))
```

扩展： csv 文件读写操作 1/2

- **csv.DictReader(** csv_file **)**
- **作用：**获取一个 csv 读取对象 _csv.dictreader。同时其获取后的数据可以使用 dict(obj) 直接转换成字典数据类型。

示例：使用 DictReader 函数读取 csv 数据并转化成字典输出(ch02-demo07-csvdictreader.py)

```
15  # 定义成函数形式
16  def getCsvData():
17      # 读取csv数据并转成字典数据类型
18      with open('CH02_Demos/data/data2.csv', 'r') as csvfile:
19          # 使用DictReader()函数读取csv数据
20          reader = csv.DictReader(csvfile)
21          print('>> reader type -> {0}'.format(reader))
22          # 返回数据
23          return [dict(line) for line in reader]
24  # 函数调用
25  print(getCsvData())
```

4.3 扩展： csv 文件读写操作 2/2

- **csv.DictWriter(** csv_file **)**
- **作用：**获取一个 csv 写入对象 _csv.dictwriter。该对象提供 writeheader( ) 完成标题写入。

示例：使用 DictWriter 函数写入 csv 数据文件（ch02-demo07-csvdictreader.py）

```python
# 写入csv数据并
data = [{'字段1':-1,'字段2':-2},{'字段1':1,'字段2':2}]

# 读取csv数据并转成字典数据类型
with open('CH02_Demos/data/data3.csv', 'w', newline='') as csvfile:
    # 获取所有字典数据的key值部分
    keys = [k for k in data[0]]
    # 使用DictWriter()函数读取csv数据
    writer = csv.DictWriter(csvfile, fieldnames=keys)
    # 调用writeheader()写入数据标题
    writer.writeheader()
    print('标题写入完毕')
    # # 使用for循环遍历数据
    for d in data:
        # 使用writewor()函数写入
        writer.writerow(d)
    print('数据写入完毕')
```

11.5 Excel 文件读写操作

① 模块介绍

② Excel 写入数据

③ Excel 读取数据

11.5.1 操作excel的模块介绍

- Python 对 Excel 操作应该是数据存储过程中最重要的格式文件之一。
- 对于 excel 文件我们并不陌生。但是 Python 要完成对 excel 文件操作需要使用 pip 下载外部模块（包）
- Python 语言本身标准模块中不包括对 excel 操作的工具模块：
    - **xlrd** 读取 excel 文件模块
    - **xlwt** 写入 excel 文件模块
- Excel 文件操作的两个重要对象 **workbook**（工作簿） 和 **sheet** （单页）

11.5.2 Excel写入操作模块xlwt模块

- **xlwt 模块** 主要完成对 excel 文件的写入操作，使用 import xlwt 前期导入后方可使用。
- **xlwt 模块** 完成 excel 文件写入的标准步骤：
    - **步骤 1**：获取 excel 文件的绝对路径
    - **步骤 2**：创建工作簿 workbook
    - **步骤 3**：在工作簿中创建 sheet 单页
    - **步骤 4**：添加数据
    - **步骤 6**：保存工作簿 workbook

11.5.3 Excel写入操作标准步骤

- **步骤 1**：获取 excel 文件的绝对路径
- **步骤 2**：创建工作簿 workbook

  函数：xlwt.**Workbook**(encoding=字符编码集) ： workbook 对象
- **步骤 3**：在工作簿中创建 sheet 单页

  函数：workbook 对象.**add_sheet**(sheet 单页的名称)：sheet 对象
- **步骤 4**：添加数据

  函数：sheet 对象.**write**(行下标 , 列下标 , 数据值)
- **步骤 5**：保存工作簿 workbook

  函数：workbook 对象.**save**(excel 文件绝对路径

11.5.4 Excel写入操作实现

示例：ch02-demo08-excel-xlwt.py

```python
# 导入xlwt模块
import xlwt
import os

# 设置文件路径
excelPath = os.path.join(os.getcwd(), 'CH02_Demos\\data\\data.xls')
# 创建excel工作簿workbook
workbook = xlwt.Workbook(encoding = 'UTF-8')
# 创建工作簿中的sheet页
sheet = workbook.add_sheet('数据测试单页')
# 设置excel标题栏内容
headers = ['字段1', '字段2', '字段3']
# 循环写入标题内容
for colIndex in range(0,len(headers)):
    # 按照字体格式样式写入标题
    sheet.write(0,colIndex,headers[colIndex])
# for循环写入数据
for rowIndex in range(1,5): # 行号
    for colIndex in range(len(headers)): # 列号
        sheet.write(rowIndex, colIndex, (rowIndex + colIndex))
# 保存创建好的excel文件
workbook.save(excelPath)
print('excel文件写入完毕')
```

11.5.4 Excel读取操作xlrd模块

示例：ch02-demo09-excel-xlrd.py

```python
# 导入xlwt模块
import xlrd
import os

# 设置文件路径
excelPath = os.path.join(os.getcwd(), 'CH02_Demos\\data\\data.xls')
# 根据excel路径打开excel文件工作簿
workbook = xlrd.open_workbook(excelPath)
# 获取sheet单页的列表
sheets = workbook.sheet_names()
# 获取第1个单页
worksheet = workbook.sheet_by_name(sheets[0])
# 外层循环获取数据行数
for i in range(0, worksheet.nrows):
    # 获取到当前行数据
    row = worksheet.row(i)
    # 内层循环控制单行的每一列
    for j in range(0, worksheet.ncols):
        # 输出当前循环到的cell坐标数据
        print(worksheet.cell_value(i, j), '\t', end=' ')
    print() # 换行显示
```

11.6本章总结

通过本章的学习，我们掌握了 Python 编程语言对常规数据文件的读写操作。

第12章：多线程

知识点

目标 1：了解线程和进程的概念及二者之间的区别

目标 2：Python 实现多线程的两种方式

目标 3：线程同步与锁机制

技能点

实战任务 1：多线程实现“统筹方法”

实战任务 2：抢票机制的模拟实现

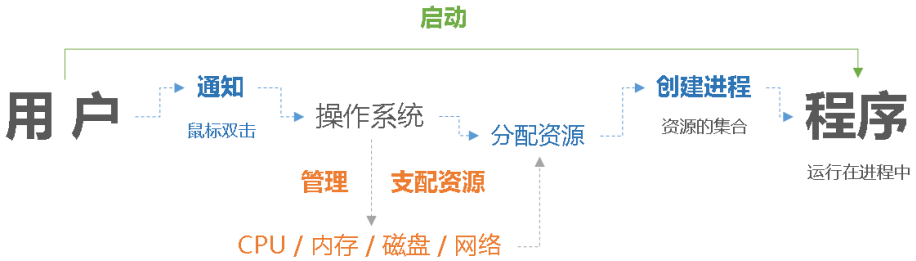
实战任务 2：“生产者消费者”设计模式的实现

12.1 多线程概述

线程的概念 / 多线程的优势 / 线程和进程的区别

12.1.1 什么是进程？

- 进程是操作系统级别的概念，它是一个应用程序运行的所需资源环境，我们称之为程序运行的上下文环境。
- 任何程序的运行都需要运行资源（如 CPU、内存、磁盘、网络……），就好比人生存需要空气、水这些资源一样。一个程序的启动，首先系统必须为该程序创建一个进程（为该程序分配一个运行环境）
- 应用程序的启动是由用户决定（操作系统启动时也会自动启动多个后台程序，以保障操作系统的运行）。
- 进程的创建是由系统执行（当用户启动应用程序时，就好比通知操作系统创建一个进程，保障应用程序有可运行的足够资源）

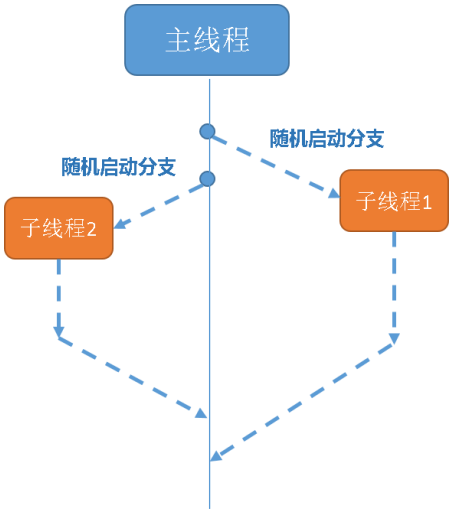


12.1.2 什么是线程？

- 线程是程序级别的概念，一个程序可以有多个线程同时执行（子功能可以同时执行）。
- 没有进程就不会有线程，线程依赖于进程。
- 线程分为程序主线程和子线程两种。
- 任何进程都会自动启动唯一的主线程。
- 主线程可以随意启动多个子线程。

- 线程是程序级别的概念，一个程序可以有多个线程同时执行（子功能可以同时执行）。
- 没有进程就不会有线程，线程依赖于进程。
- 线程分为程序主线程和子线程两种。
- 任何进程都会自动启动唯一的主线程。
- 主线程可以随意启动多个子线程。

- 线程是程序执行流的最小单元。



12.1.3 什么是多线程？

- 我们在日常使用计算机的时候，一般只会用到的 CPU **只有 30%**左右。
- 为了提高 CPU 的使用率，提高程序的运行效率，我们在编写程序的时候可以使用多线程技术。
- 所谓多线程，即让一个应用程序同时执行多个任务，是一种典型的并行执行方式，这样可以大大提高程序运行效率。
- **举例说明：**

一个银行同时开放多个窗口在办理各项业务，一个窗口就好比一个线程，窗口开得越多，办理业务的效率越高、速度越快，同时还能充分利用银行的各种资源不浪费。

如果某家银行需要提高对本市的服务质量，在各个区开设多家支行，在计算机原理中属于多进程。而一个进程就好比一个应用程序，多个进程之间又可以相互通信。

因此，在计算机中现有进程，每个进程再开启多个线程。当进程关闭的时候，当前进程（应用程序）中的多个线程会全部关闭。例如，海淀区某家支行（单独一个进程）关闭，则该银行的所有窗口（多个线程）肯定也停止服务了。

12.1.4 多线程的优势

- 多线程类似于同时执行多个不同程序（任务），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，这样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。

12.1.5 线程和进程之间的区别

- 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口和程序的出口。
- 一个 Python 脚本程序运行，Python 虚拟机就启动了一个进程，同时启动一个主线程 mainTread。
- 我们可以在一个 python 程序中创建多个子线程，每个子线程启动由程序指令控制。

12.2 Python 中的多线程实现

掌握函数及模块两种方式实现多线程

12.2.1 Python实现多线程的方式

- Python 编程语言提供了两种实现多线程的方式：

① 函数方式（涉及_thread 模块）。

② 用类来包装线程对象（涉及 threading 模块）。

12.2.2 函数方式实现多线程

- Python 编程语言提供_thread **模块**（标准模块库）
- 调用_thread **模块**中的 start_new_thread()函数来创建并启动新线程。
- **创建语法：**

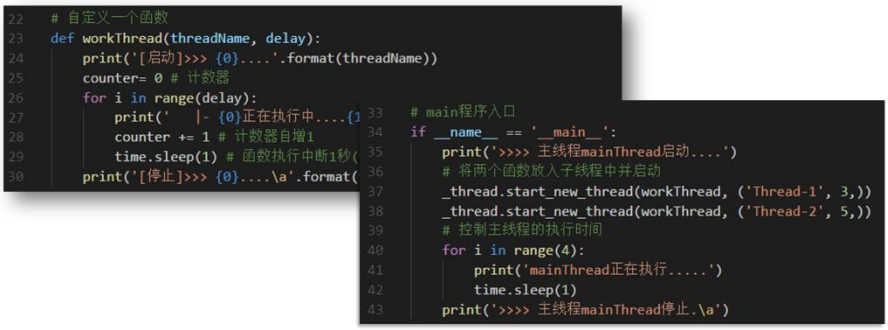
thread.start_new_thread (*function, args[, kwargs]*)

- **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。

示例：函数实现多线程 ch03-demo01-thread.py

- 下面我们一起看一下函数实现多线程。



```python
# 自定义一个函数
def workThread(threadName, delay):
    print('[启动]>>> {0}....'.format(threadName))
    counter= 0 # 计数器
    for i in range(delay):
        print('   |- {0}正在执行中....{1
        counter += 1 # 计数器自增1
        time.sleep(1) # 函数执行中断1秒(
    print('[停止]>>> {0}....\a'.format(

# main程序入口
if __name__ == '__main__':
    print('>>>> 主线程mainThread启动....')
    # 将两个函数放入子线程中并启动
    _thread.start_new_thread(workThread, ('Thread-1', 3,))
    _thread.start_new_thread(workThread, ('Thread-2', 5,))
    # 控制主线程的执行时间
    for i in range(4):
        print('mainThread正在执行.....')
        time.sleep(1)
    print('>>>> 主线程mainThread停止.\a')
```

说明：线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用 thread.exit()，他抛出 SystemExit exception，达到退出线程的目的。

12.2.3 模块实现多线程_thread

- Python 通过两个标准库_thread 和 threading 提供对线程的支持。
    - _thread 提供了低级别的、原始的线程以及一个简单的锁。
    - threading 模块提供的其他方法：

- 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 threadind.Thread 类来处理线程，threadind.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。

12.2.4 模块实现多线程的语法

- 使用模块实现多线程，首先需要导入 threading 模块，之后创建线程类，该类中必须重写 __init__ 和 run 方法。
- **线程类创建语法：**

```
# 导入线程模块
import threading
# 创建线程类
class 线程类名称(threading.Thread):
    # 重写__init__（）类初始化方法
    def __init__(self, 参数 1, ……, 参数 N）:
        # 调用父类构造方法
        threading.thread.__init__（self）:
        ……
    # 重写 run（）方法，线程启动后调用的方法
    def run（self）:
        ……具体动作……
```

- **线程类创建并启动语法：**

```
# 创建线程对象
线程对象 = 线程类名称（）
# 启动线程，自动调用线程类中的 run()方法
线程对象.start()
```

示例：模块实现多线程 ch03-demo02-threading.py

- 我们一起来看一下模块实现多线程类。

```python
# 导入多线程模块
import threading
# 导入时间模块
import time
# 创建退出标志位变量
exitFlag = 0
```

```python
# 自定义一个输出函数
def outputTime(threadName, delay, counter):
    while counter:
        if exitFlag:
            threading.Thread.exit()
        time.sleep(delay)
        print('%s: %s' % (threadName, time.ctime(time.time())))
        counter -= 1
    pass
```

- 创建线程类

```python
# 创建一个线程类
class MyThread(threading.Thread):
    # 重写构造方法
    def __init__(self, threadID, name, counter):
        # 调用父类构造方法
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    # 重写run()方法
    def run(self):
        print('启动->' + self.name)
        outputTime(self.name, self.counter, 5)
        print('结束->' + self.name)
    pass
```

```python
# 脚本程序入口
if __name__ == '__main__':
    print('mainThread主线程启动……')

    # 创建两个线程
    thread1 = MyThread(1, 'Thread-1', 1)
    thread2 = MyThread(2, 'Thread-2', 1)
    # 启动线程
    thread1.start()
    thread2.start()

    print('mianThread主线程结束.')
```

12.2.5 实战任务：多线程实现“统筹方法”

使用模块方式实现统筹方法

任务说明

- 我们都学习过“统筹方法”，假设家中来客人，需要洗茶杯和煮开水招待客人。一共需要洗 3 个茶杯，每个茶杯耗时 3 秒钟；煮开水的时间假设为 10 秒钟。两件事情并行做，

执行效率最高的方法。

- 我们将两件事情分别创建两个线程类，重写 run（）方法实现洗 3 个杯子和煮开水两个任务。
- 示例代码参见 ch03-demo03-coordinate.py

步骤1：导入相关模块

```
# 导入模块
import threading
import time
```

步骤2：创建洗杯子子线程类

```
# 创建洗杯子线程类
class WashCup(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        for i in range(1,4):
            print('开始洗第', i , '个杯子……')
            time.sleep(3)
            print('第', i , '个杯子洗完了！')
```

步骤3：创建煮开水子线程类

```
# 创建煮水子线程类
class BoilWater(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        print('开始煮水……')
        time.sleep(10)
        print('水煮好了！')
```

步骤4：创建并启动线程

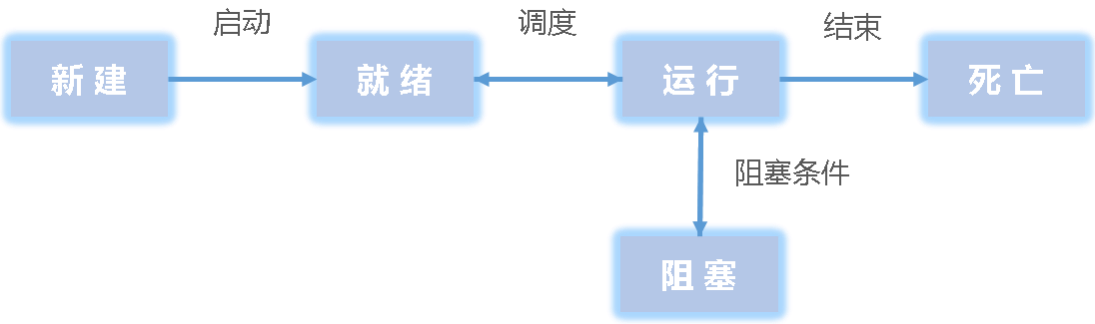
```
if __name__ == '__main__':
    # 创建线程并启动
    BoilWater().start()
    WashCup().start()
```

12.3 线程同步

掌握函数及模块两种方式实现多线程

12.3.1 线程的工作状态

- 线程有五个状态：新建、就绪、运行、死亡、阻塞
- 状态的切换如下图所示：



12.3.2 线程同步介绍

- 我们发现，多个线程在执行的时候为随机模式，有哪个线程占用 CPU 我们无法进行控制。但如果需要强行要求某一个线程执行完毕后再执行另一个线程，这种情况就需要使

用线程同步技术。

- Python 的线程同步技术有两种比较典型的解决方案：锁同步 和 条件变量同步。
- 那么，我们首先引入锁的概念。将一个线程的 run（）方法进行锁定，在全部执行完毕后在解除锁定。这样其他线程在执行的时候需要等待解除锁定后方可继续执行（锁定和解锁必须成对出现，否则出现“死锁”）。
- **创建线程锁的语法：**

```
线程锁对象 = Threading.Lock()
```

- **锁定及解锁的语法：**

```
线程锁对象.acquire() # 锁定
线程锁对象.release()  # 解除锁定
```

12.3.3 线程锁同步的使用方法

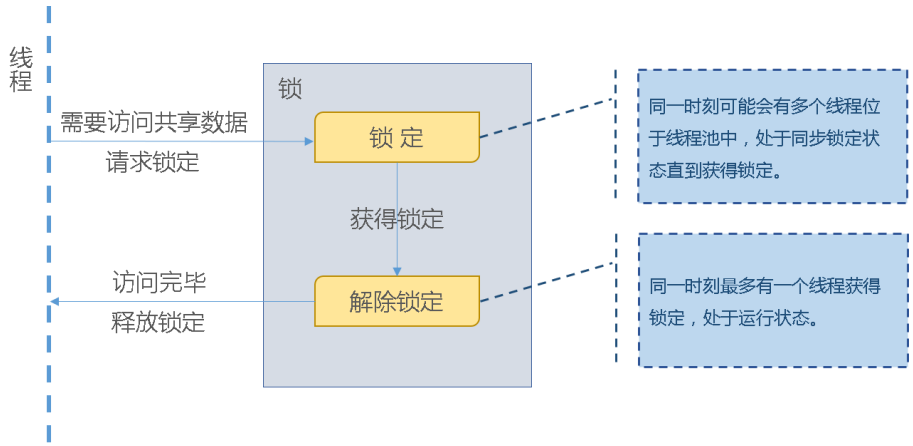
- 在我们了解了线程同步的基本知识后，在实际应用场景中我们如何使用线程同步呢？
- 我们最基本的方法其实很简单，只需要将某一个线程中的 run（）方法中的执行语句使用锁定和解除锁定两条语句包围起来即可，如下所示：
- **锁定及解锁的场景应用：**

```
……
def run(self):
    线程锁对象.acquire() # 锁定
    ……线程执行语句……
    线程锁对象.release()  # 解除锁定
```

- 这样，当该线程启动后会待线程执行语句全部执行完毕后，其他线程才会启动或继续执行……

12.3.4 线程锁同步的流程分析

- 同步锁是最 Python 中简单的线程同步方法。



示例：线程同步应用 ch04-demo04-sychronized.py

- 导入模块并创建处理函数

```python
# 导入模块
import time
import threading

# 创建线程锁
threadLock = threading.Lock()

# 创建线程列表序列
threads = []
```

```python
# 自定义函数
def print_time(threadName,delay):
    for i in range(delay):
        time.sleep(1)
        print("%s -- %s -- %s" %(threadName,i,time.ctime(time.time())))
        pass
```

示例：线程同步应用 ch04-demo04-sychronized.py

- 创建线程类并启动。

```python
# 线程类
class Mythread(threading.Thread):

    # 构造方法
    def __init__(self,threadName,delay):
        # 调用父类Thread的构造方法
        threading.Thread.__init__(self)
        self.__threadName = threadName
        self.__delay = delay

    # 线程对象启动时自动调用的run()方法
    def run(self):
        print("[启动]>>> {0}:开启线程...".format(self.__threadName))
        #获取锁用于线程同步
        threadLock.acquire()
        print('>>>> {0}锁定同步....'.format(self.__threadName))
        print_time(self.__threadName,self.__delay)
        #释放锁，开启下一个线程
        threadLock.release()
        print('>>>> {0}释放锁....'.format(self.__threadName))
        pass
```

```python
# 脚本入口
if __name__ == '__main__':
    print("[启动]>>> mainThread主线程启动....")
    #创建新线程
    thread1 = Mythread("Thread-1", 4)
    thread2 = Mythread("Thread-2", 6)

    #开启新线程
    thread1.start()
    thread2.start()

    #添加新线程到线程列表
    threads.append(thread1)
    threads.append(thread2)

    #等待所有线程完成
    for t in threads:
        t.join()

    print("[停止]>>> mainThread主线程停止....\a")
```

12.3.5 条件变量同步的介绍

- Python 提供的 Condition 对象提供了对复杂线程同步问题的支持。
- Condition 被称为**条件变量**，除了提供与 Lock 类似的 acquire 和 release 方法外，还提供了 wait 和 notify 方法。
- **条件变量同步 工作原理**
- 线程首先 acquire 一个条件变量，然后判断一些条件。如果条件不满足则 wait；如果条件满足，进行一些处理改变条件后，通过 notify 方法通知其他线程，其他处于 wait 状态的线程接到通知后会重新判断条件。不断的重复 这一过程，从而解决复杂的同步问题。

- 可以认为 **Condition** 对象维护了一个锁（Lock/RLock)和一个 waiting 池。
- 线程通过 acquire 获得 Condition 对 象，当调用 wait 方法时，线程会释放 Condition 内部的锁并进入blocked(阻塞)状态,同时在waiting池中记录这个线程。当调用notify 方法时，Condition 对象会从 waiting 池中挑选一个线程，通知其调用 acquire 方法尝试取到锁。
- **Condition** 对象的构造函数可以接受一个 Lock/RLock 对象作为参数，如果没有指定，则 **Condition** 对象会在内部自行创建一个 RLock。
- **说明:**除了 notify 方法外,Condition 对象还提供了 notifyAll 方法,可以通知 waiting 池中的所有线程尝试 acquire 内部 锁。由于上述机制，处于 waiting 状态的线程只能通过 notify 方法唤醒，所以 notifyAll 的作用在于防止有线程永远处于沉默状态。

12.4 实战任务：生产者和消费者模式

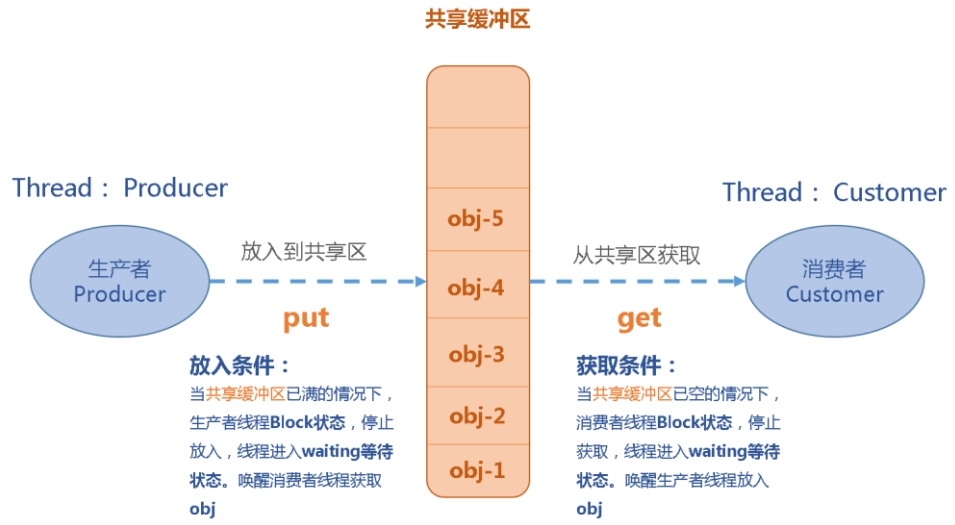
使用条件变量同步实现

12.4.1 设计模式说明

- 生产者消费者问题（英语：Producer-consumer problem），也称有限缓冲问题（英语：Bounded-buffer problem），是一个多线程同步问题的经典案例。
- 该问题描述了两个共享固定大小缓冲区的线程——即所谓的“**生产者**”和“**消费者**”——在实际运行时会发生的问题。
- **生产者**的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。

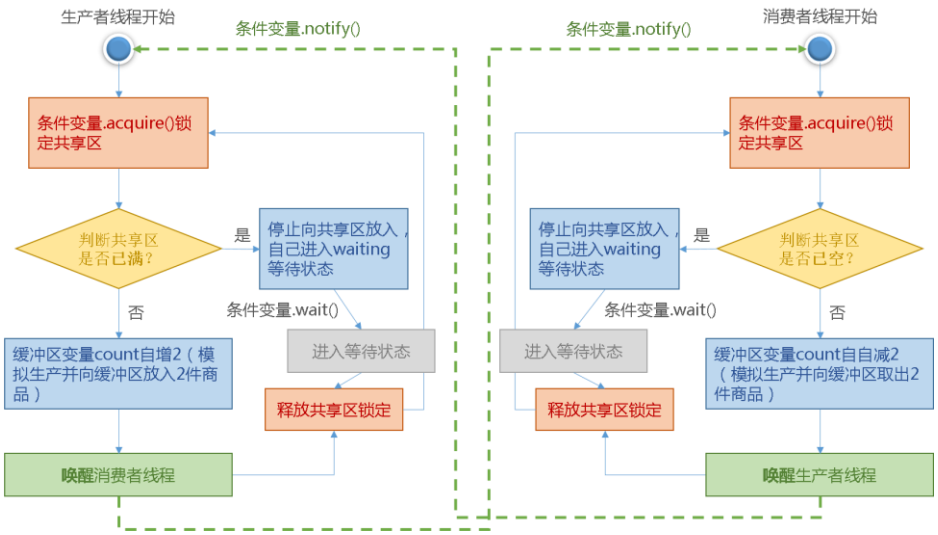
- **消费者**也在缓冲区消耗这些数据。
- 该问题的**关键**就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

设计模式示意图



12.4.2 任务分解

- 创建一个共享区，共享区的容量为 10（使用整型 count 变量模拟）
- 2 个**生产者**，每个生产者消耗随机的时间单位生产 1 件商品并放入共享区（count 自增 1 模拟），当**共享区已满**，生产者停止放入共享区，线程进入 block 阻塞状态，等待消费者线程唤醒。
- 5 个**消费者**，每个消费者使用随机的时间单位每次从共享区获取 1 件商品（count 自减 1 模拟），当**共享区已空**，消费者停止从共享区获取，线程进入 block 阻塞状态，等待生产者线程唤醒。



12.4.3 任务实现

参考代码：ch04-demo06-customer.py

Step1：导入相关模块，创建共享区变量及条件对象

```python
#-*- coding:utf-8 -*-
'''
    ch04-demo06-customer.py
    ------------------------
    生产者-消费者

    @Copyright: Chinasoft International·ETC
    @author: Alvin
    @date: 2018-04-24
'''

# 导入多线程模块
import threading
# 导入时间模块
import time
# 导入随机模块
import random

# 使用共享区模拟变量
count = 0
# 创建条件对象
condition = threading.Condition()
```

Step2：创建 生产者 线程类

```python
# 生产者线程类
class Producer(threading.Thread):
    # 重写构造方法
    def __init__(self, threadName):
        threading.Thread.__init__(self)
        self.threadName = threadName
    # 重写run()方法
    def run(self):
        global count  # 引用全局共享变量count
        while True:
            # 使用条件对象获取锁并锁定
            if condition.acquire():
                # 判断共享变量是否已达到上限(已满)
                if count >= 10:
                    print('共享区以满，生产者Producer线程进入阻塞Block状态，停止放入！')
                    condition.wait() # 当前线程进入到阻塞状态
                else:
                    count += 1  # 共享变量自增1
                    msg = time.ctime() + ' ' + self.threadName + '生产了 1 件商品放入共享区，共享区总计商品个数:
                    print(msg)
                    condition.notify()  # 唤醒其他阻塞状态的线程(如，消费者线程)
                condition.release()  # 解除锁定
                time.sleep(random.randrange(10)/5) # 随机休眠N秒
```

Step3：创建 消费者 线程类

```python
# 消费者线程类
class Customer(threading.Thread):
    # 重写构造方法
    def __init__(self, threadName):
        threading.Thread.__init__(self)
        self.threadName = threadName
    # 重写run()方法
    def run(self):
        global count # 引用全局共享变量count
        while True:
            # 使用条件对象获取锁并锁定
            if condition.acquire():
                # 判断共享变量是否已为0(已空)
                if count < 1:
                    print('共享区以空，消费者Customer线程进入阻塞Block状态，停止获取！')
                    condition.wait()  # 当前线程进入到阻塞状态
                else:
                    count -= 1  # 共享变量自减1
                    msg = time.ctime() + ' ' + self.threadName + '消费了 1 件商品，共享区总计商品个数：  ' + str(count)
                    print(msg)
                    condition.notify()  # 唤醒其他阻塞状态的线程(如，生产者线程)
                condition.release()  # 解除锁定
                time.sleep(random.randrange(10)) # 随机休眠N秒
```

Step4：创建 2 个生产者和 5 个消费者

```
# 脚本程序入口
if __name__ == '__main__':
    for i in range(2):
        p = Producer('[生产者-'+ str(i+1) +']')
        p.start()
    for i in range(5):
        c = Customer('消费者-'+ str(i+1) +']')
        c.start()
```

```
Thu Apr 26 13:40:09 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 1
Thu Apr 26 13:40:09 2018 [生产者-2]生产了 1 件商品放入共享区，共享区总计商品个数： 2
Thu Apr 26 13:40:09 2018 消费者-1]消费了 1 件商品，共享区总计商品个数： 1
Thu Apr 26 13:40:09 2018 消费者-2]消费了 1 件商品，共享区总计商品个数： 0
共享区以空，消费者Customer线程进入阻塞Block状态，停止获取！
共享区以空，消费者Customer线程进入阻塞Block状态，停止获取！
共享区以空，消费者Customer线程进入阻塞Block状态，停止获取！
Thu Apr 26 13:40:09 2018 [生产者-2]生产了 1 件商品放入共享区，共享区总计商品个数： 1
Thu Apr 26 13:40:09 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 2
Thu Apr 26 13:40:09 2018 消费者-4]消费了 1 件商品，共享区总计商品个数： 1
Thu Apr 26 13:40:10 2018 消费者-2]消费了 1 件商品，共享区总计商品个数： 0
Thu Apr 26 13:40:10 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 1
Thu Apr 26 13:40:10 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 2
Thu Apr 26 13:40:11 2018 [生产者-2]生产了 1 件商品放入共享区，共享区总计商品个数： 3
Thu Apr 26 13:40:11 2018 消费者-1]消费了 1 件商品，共享区总计商品个数： 2
Thu Apr 26 13:40:11 2018 消费者-1]消费了 1 件商品，共享区总计商品个数： 1
Thu Apr 26 13:40:12 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 2
Thu Apr 26 13:40:12 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 3
Thu Apr 26 13:40:12 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 4
Thu Apr 26 13:40:12 2018 [生产者-2]生产了 1 件商品放入共享区，共享区总计商品个数： 5
Thu Apr 26 13:40:13 2018 [生产者-1]生产了 1 件商品放入共享区，共享区总计商品个数： 6
Thu Apr 26 13:40:13 2018 [生产者-2]生产了 1 件商品放入共享区，共享区总计商品个数： 7
Thu Apr 26 13:40:13 2018 [生产者-2]生产了 1 件商品放入共享区，共享区总计商品个数： 8
Thu Apr 26 13:40:13 2018 消费者-1]消费了 1 件商品，共享区总计商品个数： 7
```

12.5 本章总结

通过本章节的学习我们了解的进程和线程的概念，同时掌握了多线程编程的两种典型应用方式。对 Python 编程语言在多线程处理方面打下了坚实的基础。同时深入探讨了了线程安全问题以及在实际开发中典型的应用模型。