

中国大学 MOOC《零基础学 Python 语言》*

纸上得来终觉浅，绝知此事要躬行。

——《冬夜读书示子聿》 陆游

万震
重庆大学

2017 年 12 月 10 日

*文件版本号 v1.0，本笔记仅供个人学习使用

目 录

1	课程介绍	1
1.1	课程导言	1
1.2	课程内容	1
1.3	参考资料	2
1.4	学习建议	2
2	Python 编程之基本方法	2
2.1	计算机的概念	2
2.1.1	计算机分类依据	3
2.1.2	程序设计语言的重要性	4
2.2	程序设计语言概述	4
2.2.1	程序设计语言种类	4
2.2.2	编译和解释	5
2.2.3	脚本语言	5
2.3	Python 语言介绍	6
2.4	Python 开发环境配置	6
2.4.1	安装	6
2.4.2	配置	7
2.4.3	启动	8
2.4.4	第一个 Python 程序	11
2.5	程序设计的基本方法	11
2.6	理解问题的计算部分	12
2.7	温度转换程序实例	13
2.8	课后练习	14
3	Python 编程之实例解析	15
3.1	Python 语法元素分析	15
3.1.1	缩进	15
3.1.2	注释	15
3.1.3	常量与变量	16
3.1.4	命名	16
3.1.5	表达式	17
3.1.6	输入函数	18
3.1.7	分支语句	18
3.1.8	赋值语句	18
3.1.9	输出函数	19

3.1.10	循环语句	19
3.2	程序编写模板	20
3.2.1	input-print 模板	20
3.2.2	initial-print 模板	20
3.3	turtle 库与蟒蛇绘制程序	20
3.3.1	蟒蛇绘制程序分析	22
3.3.2	Turtle 库快速参考	23
3.4	函数库的引用	24
4	Python 编程之数据类型	25
4.1	Python 中类型的概念	25
4.1.1	数字类型	25
4.1.2	字符串类型	28
4.1.3	元组类型	34
4.1.4	列表类型	35
4.2	math 库和 random 库	37
4.2.1	math 库	37
4.2.2	random 库	38
4.2.3	π 的计算	40
5	Python 编程之控制结构	42
5.1	程序基本结构	42
5.1.1	顺序结构	42
5.1.2	选择结构	42
5.1.3	循环结构	43
5.2	简单分支	43
5.3	多分支	45
5.4	异常处理	46
5.5	三者最大实例分析	48
5.6	基本循环结构	50
5.6.1	for 循环	50
5.6.2	while 循环	51
5.6.3	循环中的 break,continue 和 else	52
5.7	通用循环构造方法	54
5.7.1	交互式循环	54
5.7.2	哨兵循环	55
5.7.3	文件循环	55
5.8	死循环、后测循环和半路循环	56

5.8.1	死循环	56
5.8.2	后测循环	56
5.8.3	半路循环	57
5.9	布尔表达式	57
5.9.1	布尔操作符	58
5.9.2	布尔代数	58
6	Python 编程之代码复用	61
6.1	函数	61
6.1.1	函数的定义	61
6.1.2	函数的调用和返回	63
6.1.3	改变参数值的函数	65
6.2	函数与递归	66
6.2.1	函数和程序结构	66
6.2.2	递归	69
6.3	函数实例分析	71
7	Python 编程之组合类型	74
7.1	文件	74
7.1.1	文件的基础	74
7.1.2	文件的基本处理	74
7.1.3	文件实例一	78
7.1.4	文件实例二	80
7.2	字典	82
7.2.1	字典的基础	83
7.2.2	字典的操作	85
7.2.3	字典实例一	89
7.2.4	字典实例二	93
8	Python 编程之计算生态	96
8.1	程序设计方法	96
8.1.1	计算思维	96
8.1.2	自顶向下的设计	96
8.1.3	自底向上的执行	102
8.2	软件开发方法基础	105
8.2.1	软件开发方法	105
8.2.2	敏捷开发方法	107
8.3	面向过程程序设计	108
8.4	面向对象程序设计	111

8.5	面向对象实例	114
8.6	面向对象的特点	115
8.6.1	封装	115
8.6.2	多态	115
8.6.3	继承	116
参考文献		117

1 课程介绍

1.1 课程导言

计算机是计算工具，更是创新平台，高效有趣地利用计算机需要更简洁实用的编程语言。Python 语言，由 Guido van Rossum 大牛在 1989 年发明，它是当今世界最受欢迎的计算机编程语言，也是一门“学了有用、学了能用、学会能久用”的计算生态语言。本课程面向编程零基础同学，以兴趣为驱动，学习并实践 Python 语言，“轻松编程、享受创新”^[1]。

Python 的流行

1.2 课程内容

本课程共有 3 个教学单元，共 6 周，教学安排如下：

课程大纲

单元 1: Python 快速入门 (2 周)

第 1 周：基本程序设计

教学内容：计算机的概念、程序设计语言类型（编译型、解释型）、程序设计语言种类、Python 语言初见、Python 语言开发环境配置、基本的程序设计方法 IPO

第 2 周：Python 程序入门

教学内容：Python 程序设计实例剖析、Python 语言元素：程序框架、注释、常量、变量、表达式、输入输出、赋值、分支、循环、函数等、结合 Turtle 库的图形输出编程实例剖析、程序设计模板。

单元 2: Python 语言语法 (2 周)

第 3 周：类型及应用、程序控制结构

教学内容：类型的概念、数字类型、数学函数的使用、字符串类型、字符串的各种处理方法、元组类型、列表类型、列表的各种使用方法。

第 4 周：函数和递归

教学内容：函数、函数调用方法、函数返回值、函数与程序结构、递归及使用。

单元 3: 程序设计方法 (2 周)

第 5 周：交互式图形编程

教学内容：程序设计方法学、图形对象概念、交互式图形用户接口、图形库应用方法、turtle 库。

第 6 周：Python 图形艺术

教学内容：turtle 库的使用、图形艺术。

1.3 参考资料

Python 集成开发环境 (IDE)

相关资料

[1] IDLE: Python 解释器默认工具（推荐）

[2] [Anaconda](#)

[3] [PyCharm](#)

参考网站

[1] [Python 主站](#)

[2] [Python Beautiful Soup](#)

[3] [Python Scrapy](#)

1.4 学习建议

学习建议

- 跟上进度：跟随课程进度，完成课程要求的学习内容
- 重视练习：请课后进行额外程序设计练习
- ♠ 每周课后用 2 个小时进行练习，熟能生巧

2 Python 编程之基本方法

2.1 计算机的概念

Computer，原指专门负责计算的人，后来演变成特指计算设备，译为“计算机”。

定义 2.1 (计算机) 计算机是能够根据一组指令操作数据的机器。

A computer is a machine that manipulates data according to a list of instructions.

定义 2.2 (存储程序) ^① 存储程序包含三个基本含义：

- 计算机（指硬件）由运算器、控制器、存储器、输入设备和输出设备等五大基本部件组成
- 计算机内部采用二进制来表示指令和数据
- 将编写好的程序和原始数据事先存入存储器，然后再启动计算机工作

^①存储程序结构概念由美籍匈牙利科学家冯 诺依曼等人于 1946 年提出，也叫冯诺依曼结构

输入设备和输出设备，是指计算机从外界获得信息或将结果返回的装置，五大部件对应硬件：

- 中央处理器（CPU），控制器和运算器
- 存储器，主存储器（内存）和辅助存储器（硬盘）
- 外部设备（输入输出设备）

计算机的工作过程：

- 程序：编写好程序放到存储器中
- 数据：所用到的数据放到存储器中
- 计算：计算机从存储器某些位置取数据并计算，然后将数据存储在某些位置
- 停机：程序执行后自动停机

2.1.1 计算机分类依据

计算机分类依据：运算速度、成本、机器尺寸、复杂性、应用背景等。

微型计算机 主机的硬件系统：

- 微处理器：系统的计算核心，对应运算器和控制器
- 内存和硬盘：存储数据的地方，对应存储器
- I/O 接口：计算机与外设进行信息交换的“桥梁”，对应输入和输出设备
- 总线：以主板为载体，连接上述部分

嵌入式计算机 最贴近我们生活的一类计算机，完全嵌入受控器件内部，为特定应用而设计的专用计算机；

运行在资源有限的计算机硬件，内存较小，没有键盘，甚至没有屏幕。

嵌入式计算机属于程序存储计算机。

超级计算机 在计算速度或容量上领先世界的电子计算机；

具有鲜明的时代特点；

体系设计和运作机制与个人计算机有很大区别。

世界超级计算机排行榜：**TOP 500**，世界超级计算机排行榜每年两次选出世界上最快的 500 台计算机，是国家科技实力的重要体现。

超级计算机常用于需要大量运算的工作，如天气预测、气候研究、运算化学、分子模型、物理模拟、密码分析、汽车设计、生物信息、挑战人类等；

超级计算机由需求产生，服务于科学进步；

超级计算机的设计理念影响着其他类型计算机的发展。

服务器级计算机 一种高性能计算机，从性能上介于微机和超级计算机之间。

运行一类管理资源并为用户提供 7*24 服务的计算机软件。

文件服务器、数据库服务器、邮件服务器、邮件服务器、域名服务器等一系列功能的主要计算载体。

网络专用计算机 网络专用计算机指计算机网络所使用的专用计算机设备。主要功能包括：路由器、交换机、防火墙、网络入侵检测设备等。

工业控制计算机 采用现代大规模集成电路技术，严格的生产工艺制造，内部电路采取了抗干扰技术，具有很高的可靠性，例如，电梯控制、汽车中控锁等。

广泛应用于钢铁、石油、化工、电力、机械制造、汽车、轻纺、交通运输、环保等各个行业。

传感器节点计算机

定义 2.3 (传感器) 传感器是一种以测量为目的，以一定精度把被测量转换为易于处理的电量信号输出的装置。

传感器节点计算机是传感器与小型计算机的结合，为推动人类感知地球的技术进程（物联网）做出贡献。

2.1.2 程序设计语言的重要性

程序设计是展示计算机强大能力的主要手段，无论利用何种计算机，都需要学习：程序设计语言。

编程语言学习
的重要性

让我们正式开启程序设计之旅……

2.2 程序设计语言概述

计算机是能够根据一组指令操作数据的机器，它有两个特性：

- 功能性：可以进行数据计算
- 可编程性：根据一系列指令来执行

计算机的可编程性需要通过程序设计来体现。

程序设计语言，也叫编程语言，是计算机能够理解和识别操作的一种交互体系。最好的程序设计语言是人类的自然语言。

自然语言存在的问题：存在表达歧义；文学色彩浓厚。因此，还无法借助自然语言进行程序设计。

2.2.1 程序设计语言种类

- 机器语言：01 代码，CPU 认识的语言

例 2.1 2+3 的运算 1101001000111011

- 汇编语言：在机器语言上增加了人类可读的助记符

例 2.2 2+3 的运算 add 2, 3, result

- 高级语言：向自然语言靠近的语言

例 2.3 2+3 的运算 `result = 2 + 3`

历史上出现过 600 多种程序设计语言，这些语言的名字覆盖字母 A 到 Z。常用的程序设计语言：100 余种，C、C++、VB、Java、JavaScript、Ruby、Swift、Python、Verilog、VHDL、PHP/HTML 等。

2.2.2 编译和解释

- 编译：将高级语言源代码转换成目标代码（机器语言），程序便可以运行，如图 2-1 所示。

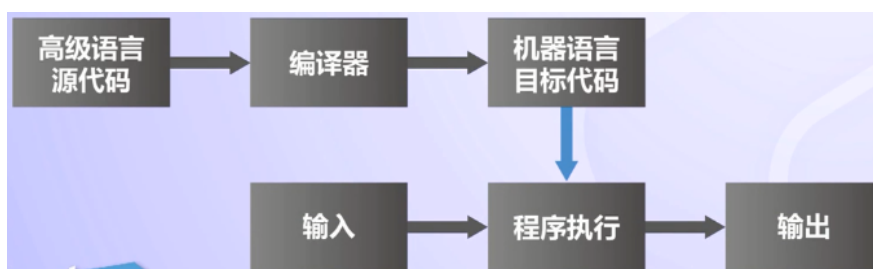


图 2-1 编译示意图

- 解释：将高级语言源代码逐条转换成目标代码同时逐条执行，每次运行程序需要源代码和解释器，如图 2-2 所示。

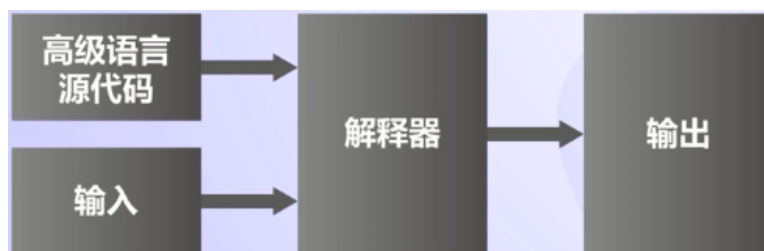


图 2-2 解释示意图

编译的好处：

- 目标代码执行速度更快
- 目标代码在相同操作系统上使用灵活

解释的好处：

- 便于维护源代码
- 良好的跨平台可移植性

2.2.3 脚本语言

- 静态语言：编译执行的编程语言，如 C、Java 等

- 脚本语言：解释执行的编程语言，如 PHP、JavaScript 等
 - ♠ Python 语言是脚本语言

2.3 Python 语言介绍

Python, 译为“蟒蛇”, Python 语言的拥有者是 **Python Software Foundation** (PSF), PSF 是非盈利组织, 致力于保护 Python 语言开放、开源和发展。

Python 语言历史:

- 2000 年 10 月, Python 2.0
- 2008 年 12 月, Python 3.0

Python 语言的版本更迭

- 更高级别的 3.0 系列不兼容早期 2.0 系列
- 2008 年至今, 版本更迭带来大量库函数的升级替换, Python 语言的版本更迭痛苦且漫长
- 到今天, Python 3.x 系列已经成为主流

Python 语言特点: 通用语言、是脚本语言、开源语言、跨平台语言、多模型语言。

2.4 Python 开发环境配置

2.4.1 安装

Python 主页: <https://www.python.org/downloads/>, 下载并安装 Python 基本开发和运行环境, 根据操作系统不同选择不同版本下载相应的 Python 3.0 系列版本程序, 如图 2-3 所示。安装时, 勾选将 python 添加进环境变量。

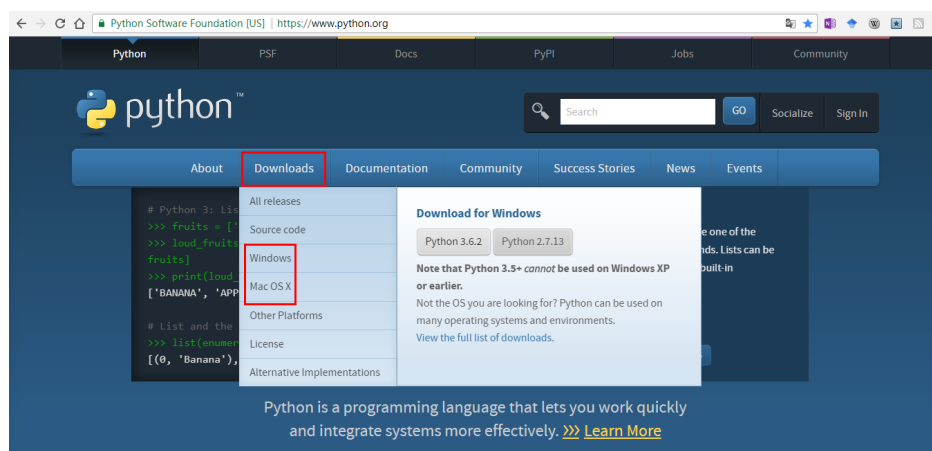


图 2-3 Python 下载页

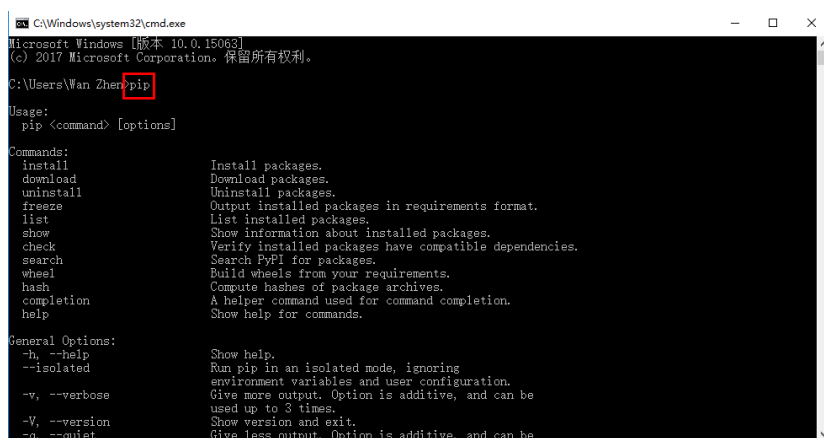
2.4.2 配置

Python 的包管理工具 pip 的安装在本版本 Python 2 \geq 2.7.9 或 Python 3 \geq 3.4 安装过程中已经完成, 无需另外安装, 若安装的 python 版本不介于二者, 安装 pip 方法如下^①:

1 下载 get-pip.py, 将 <https://bootstrap.pypa.io/get-pip.py> 中内容复制粘贴到 txt 中, 保存后修改其缀名为.py;

2 在 cmd 中依次执行命令 > python get-pip.py
> pip install atx

包的安装方法: Win+R 启动“运行”命令框, 输入 cmd 后回车, 在命令行中输入: pip, 回车, 验证 pip 是否添加到环境变量中, 如图 2-4 所示。



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.15063]
(c) 2017 Microsoft Corporation. 保留所有权利。

C:\Users\Wan Zhen>pip

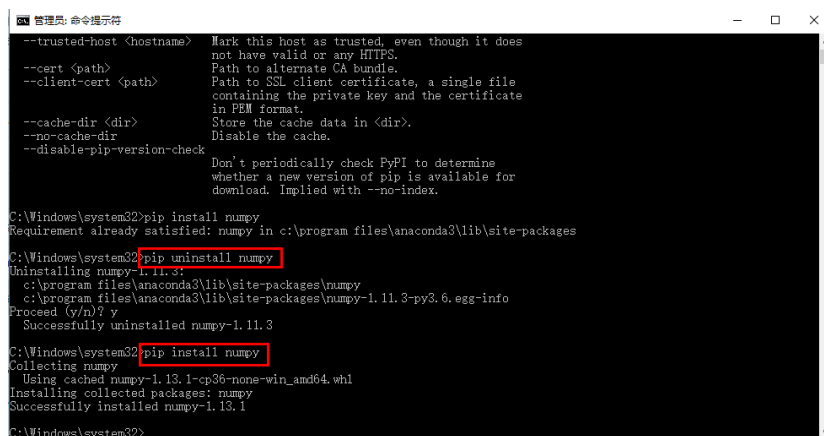
Usage:
  pip <command> [options]

Commands:
  install             Install packages.
  download            Download packages.
  uninstall           Uninstall packages.
  freeze              Output installed packages in requirements format.
  list                List installed packages.
  show                Show information about installed packages.
  check               Verify installed packages have compatible dependencies.
  search              Search PyPI for packages.
  wheel               Build wheels from your requirements.
  hash                Compute hashes of package archives.
  completion          A helper command used for command completion.
  help                Show help for commands.

General Options:
  -h, --help            Show help.
  --isolated             Run pip in an isolated mode, ignoring
                        environment variables and user configuration.
  -v, --verbose          Give more output. Option is additive, and can be
                        used up to 3 times.
  -V, --version          Show version and exit.
  -q, --quiet            Give less output. Option is additive, and can be
```

图 2-4 验证 pip 是否添加到环境变量

安装包的命令格式为: pip <command> [options]; 若卸载某个包, 输入 pip uninstall [packagename] 即可, 如图 2-5 所示。



```
管理壳: 命令提示符
--trusted-host <hostname> Mark this host as trusted, even though it does
                        not have valid or any HTTPS.
--cert <path>            Path to alternate CA bundle.
--client-cert <path>     Path to SSL client certificate, a single file
                        containing the private key and the certificate
                        in PEM format.
--cache-dir <dir>        Store the cache data in <dir>.
--no-cache-dir            Disable the cache.
--disable-pip-version-check
                        Don't periodically check PyPI to determine
                        whether a new version of pip is available for
                        download. Implied with --no-index.

C:\Windows\system32>pip install numpy
Requirement already satisfied: numpy in c:\program files\anaconda3\lib\site-packages

C:\Windows\system32>pip uninstall numpy
Uninstalling numpy-1.11.3:
  c:\program files\anaconda3\lib\site-packages\numpy
  c:\program files\anaconda3\lib\site-packages\numpy-1.11.3-py3.6.egg-info
Proceed (y/n)? y
  Successfully uninstalled numpy-1.11.3

C:\Windows\system32>pip install numpy
Collecting numpy
  Using cached numpy-1.13.1-cp36-none-win_and64.whl
Installing collected packages: numpy
Successfully installed numpy-1.13.1

C:\Windows\system32>
```

图 2-5 卸载包

^①<https://pip.pypa.io/en/latest/installing/>

2.4.3 启动

方法 1: Win+R 启动“运行”命令框, 输入 python 后回车, 如图 2-6 所示。

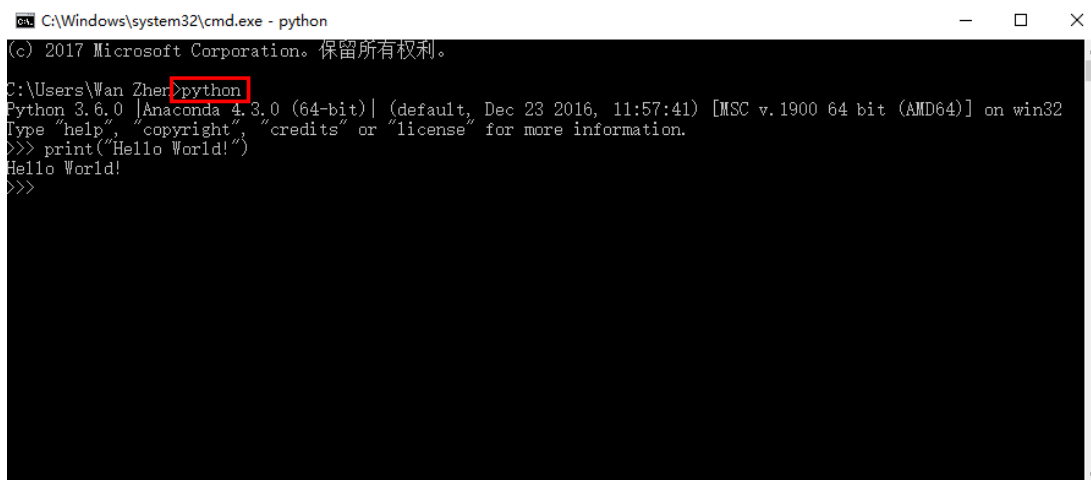


图 2-6 启动方法 1

方法 2: 调用 IDLE 来启动 Python 图形化运行环境 (Shell 方式), Shell 是交互式的解释器; 输入一行命令, 解释器就解释运行, 行出相应结果。如图 2-7 所示。

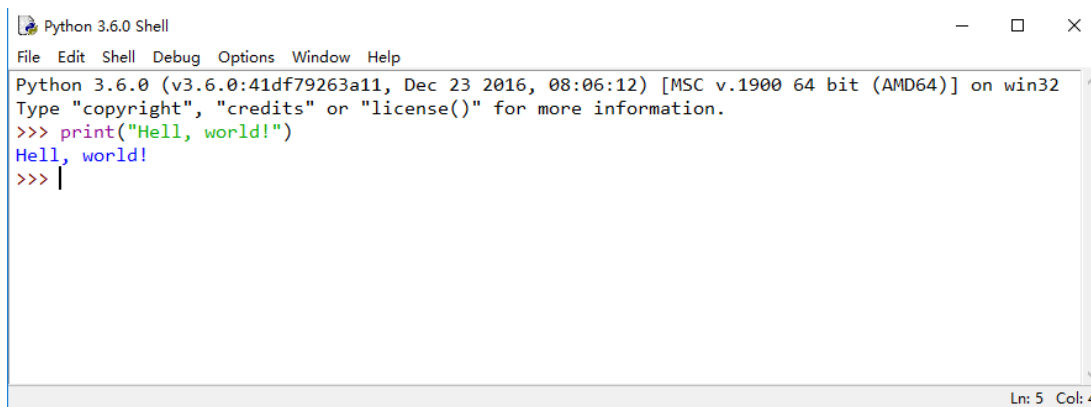


图 2-7 启动方法 2

方法 3: 按照语法格式编写代码, 编写可以用任何文本编辑器, 保存为 *.py 文件 (文件方式)。如图 2-8 所示。

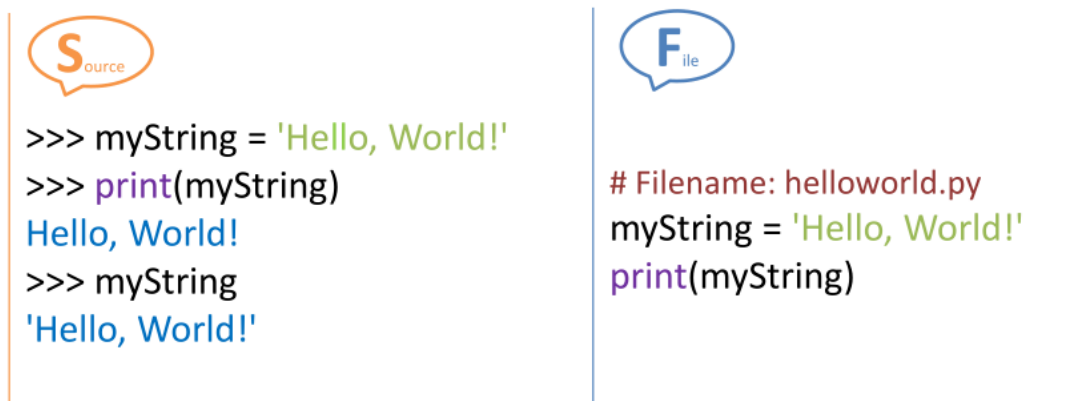


图 2-8 启动方法 3

方法 4: 将 Python 集成到 Anaconda、Eclipse、PyCharm 等面向较大规模项目开发的集成开发环境中。

以 Anaconda 为例，去 <https://www.continuum.io/downloads>，根据不同的 Python 版本下载相应的 Anaconda，如图 2-9 所示。

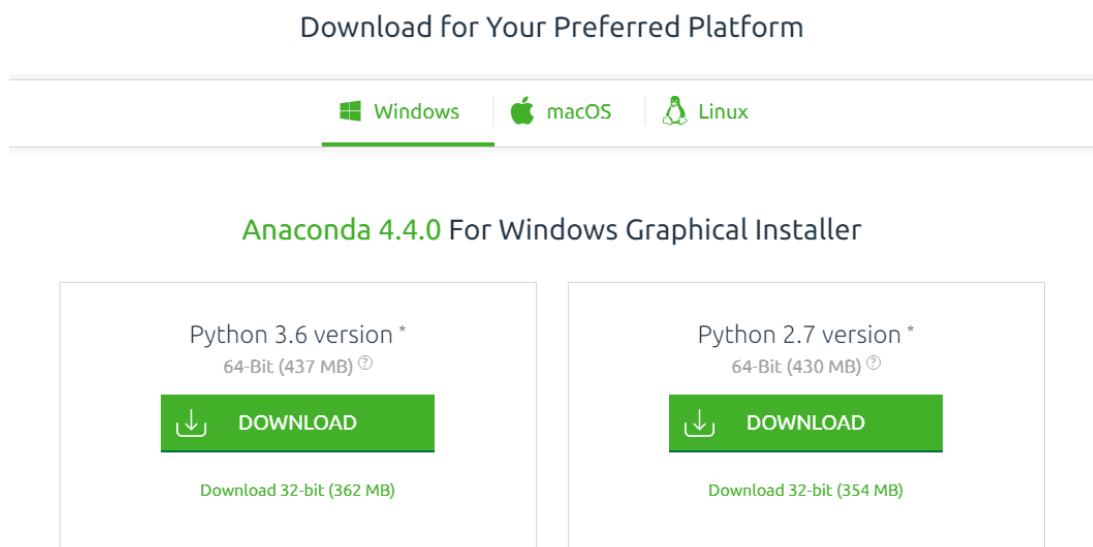


图 2-9 Anaconda

双击“Anaconda3-4.4.0-Windows-x86_64.exe”(或“Anaconda2-4.4.0-Windows-x86_64.exe”)进入安装向导，在第四步时，勾选将 python3.6 添加进环境变量，这样即可在 cmd 中启动 spyder。如图 2-10 所示。

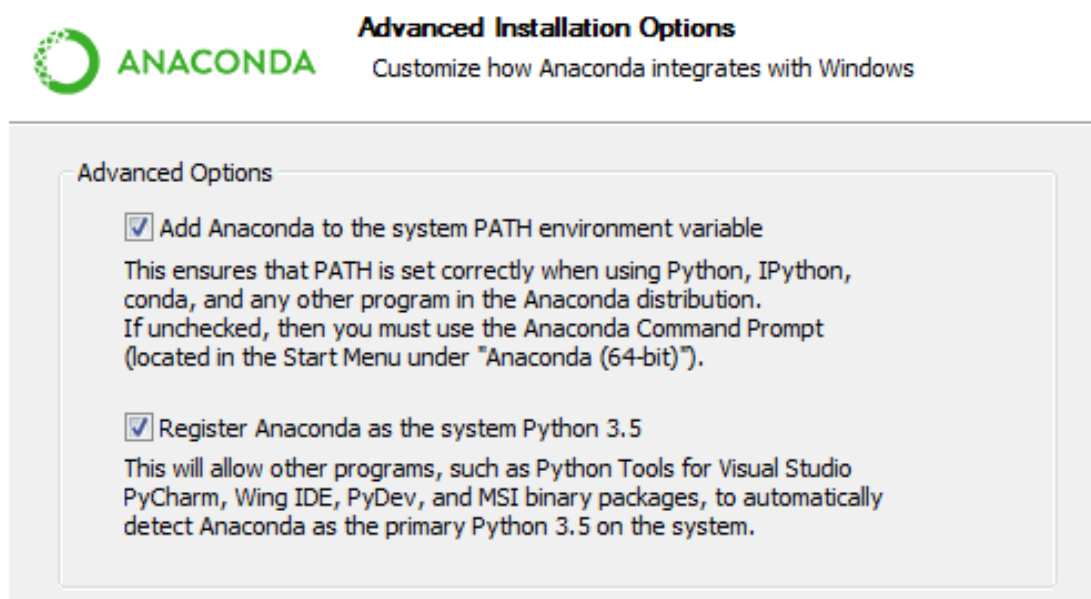


图 2-10 将 python3.6 添加进系统环境变量

安装完成后，Win+R 启动“运行”命令框，输入 spyder 回车，即可打开，如图 2-11 所示。

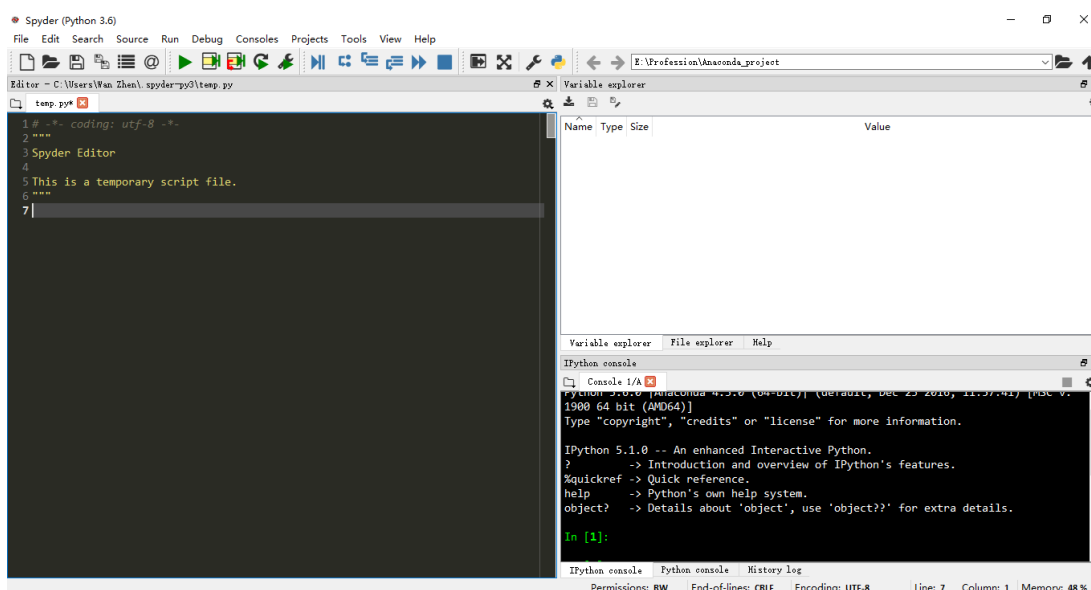


图 2-11 Spyder 界面

Anaconda 利用工具/命令 conda 来进行 package 和 environment 的管理，并且已经包含了 Python 和相关的配套工具。conda 的核心功能是包管理与环境管理，包管理与 pip 的使用类似；环境管理则允许用户方便地安装不同版本的 python 并可以快速切换。Anaconda 则是一个打包的集合，里面预装了 conda、某个版本的 python、众多 packages、科学计算工具等，conda 将几乎所有的工具、第三方包都当做 package 对待，甚至包括 python 和 conda 自身，因此，

conda 打破了包管理与环境管理的约束，能非常方便地安装各种版本 python、各种 package 并方便地切换。

Conda 的包管理与 pip 类似，常用操作如下：

- 1 conda list 查看当前环境下已安装的包
- 2 conda install packagename 安装包
- 3 conda search packagename 查找 package 信息
- 4 conda update packagename 更新包
- 5 conda remove packagename 删除包
- 6 conda update conda 更新 conda，保持 conda 最新
- 7 conda update anaconda 更新 anaconda
- 8 conda update python 更新 python

如果需要安装很多 packages，我们会发现 conda 下载的速度经常很慢，因为 Anaconda.org 的服务器在国外。所幸的是，清华 TUNA 镜像源有 Anaconda 仓库的镜像，我们将其加入 conda 的配置即可：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
```

执行完上述命令后，会生成 C:\Users\USER_NAME\.condarc 文件，记录着我们对 conda 的配置，直接手动创建、编辑该文件是相同的效果。

2.4.4 第一个 Python 程序

Python 输出函数：print()，输出的可以是变量或者字符串。

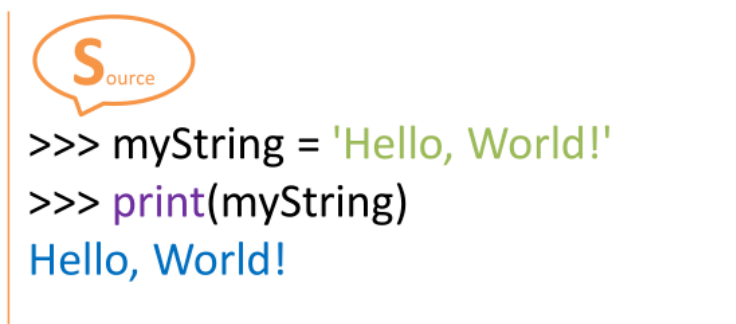


图 2-12 输出 Hello, World

2.5 程序设计的基本方法

IPO 模式

- I: Input 输入，程序的输入。包括：文件输入、网络输入、用户手工输入、随机数据输入、程序内部参数输入等，输入是一个程序的开始

- **P: Process** 处理，程序的主要逻辑。程序对输入进行处理，输出产生结果；处理的方法也叫算法，是程序最重要的部分；**算法是一个程序的灵魂**
- **O: Output** 输出，程序的输出。程序的输出包括：屏幕显示输出、文件输出、网络输出、操作系统内部变量输出等，输出是一个程序展示运算成果的方式

♠ 死循环

```
while True:  
    a=1
```

它是没有输入输出的程序。“死循环”也有价值，它通过不间断执行，快速消耗 CPU 的计算资源，可以用来测试 CPU 性能。

2.6 理解问题的计算部分

问题的计算部分指一个待解决问题中，可以用程序辅助完成的部分

实际问题抽象
为数学模型，
即算法



图 2-13 问题的计算特性

例 2.4 对于去美国旅行的中国游客来说，会遇到一个苦恼的问题：美国地区的温度采用华氏温度度量，而我国居民更为习惯使用摄氏温度，因此，在美国无论天气预报还是空调调节，中国旅客都很难习惯。相反，来中国旅游的美国游客，也有温度习惯不同带来的烦恼。

问题的计算部分：通过计算技术解决温度转换

- 1 根据两种温度的换算公式，写一个温度转换程序，由人把温度输入，程序将转换后温度输出
- 2 网络上有类似的在线程序，写一个网络程序，把人输入的温度发送到互联网上，获得转换结果后输出
- 3 写一个程序，可以通过 GPS 定位，获得使用者当前的位置，再通过网络获取当前位置的温度信息，自动进行转换（不需要使用者输入）
- 4 写一个程序，可以通过程序监听温度信息发布渠道，比如收音机、电视等，通过语音识别、图像识别等方法自动获得温度信息的数值，自动完成转换

程序编写的步骤:

- A 分析问题: 分析问题的计算部分
- B 确定问题: 将计算部分划分为确定的IPO 三部分
- C 设计算法: 完成计算部分的核心方法
- D 编写程序: 实现整个程序
- E 调试测试: 使程序在各种情况下都能正确运行
- F 升级维护: 使程序长期正确运行, 适应需求的微小变化

2.7 温度转换程序实例

温度刻画存在不同体系, 摄氏度以1标准大气压下水的结冰点为0度, 沸点为100度, 将温度进行等分刻画。华氏度以1标准大气压下水的结冰点为32度, 沸点为212度, 将温度进行等分刻画。

问题: 如何利用 Python 程序进行摄氏度和华氏度之间的转换?

步骤 1: 分析问题的计算部分: 采用公式转换方式解决计算问题

步骤 2: 确定功能

- 输入: 华氏或者摄氏温度值、温度标识
- 处理: 温度转化算法
- 输出: 华氏或者摄氏温度值、温度标识 (F 表示华氏度, C 表示摄氏度)

步骤 3: 设计算法 根据华氏和摄氏温度定义, 转换公式如下:

$$C = (F - 32) / 1.8 \quad F = C * 1.8 + 32$$

步骤 4: 编写程序

```
1  # TempConvert.py
2  val = input(" 请输入带温度表示符号的温度值 (例如: 32C): ")
3  if val[-1] in ['C', 'c']:
4      f = 1.8 * float(val[0:-1]) + 32
5      print(" 转换后的温度为: %.2fF"%f)
6  elif val[-1] in ['F', 'f']:
7      c = (float(val[0:-1]) - 32) / 1.8
8      print(" 转换后的温度为: %.2fC"%c)
9  else:
10     print(" 输入有误")
```

步骤 5: 调试、运行程序 使用 IDLE 打开上述文件, 按 F5 运行 (推荐), 输入数值, 观察输出

步骤 6: 升级维护

步骤简化

编写程序至少需要 3 个步骤：确定 IPO、编写程序、调试程序。

2.8 课后练习

例 2.5 整数序列求和。用户输入一个正整数 N，计算从 1 到 N（包含 1 和 N）相加之后的结果。

```
1 n = input(" 请输入整数 N: ")
2 sum = 0
3 for i in range(int(n)):
4     sum += i + 1
5 print("1 到 N 求和结果: ", sum)
```

例 2.6 阶乘计算。计算 $1+2!+3!+\dots+10!$

```
1 sum, tmp = 0, 1
2 for i in range(1,11):
3     tmp*=i
4     sum+=tmp
5 print(" 运算结果是: {}".format(sum))
```

熟练使用
Python 的
IDLE 编程环境

例 2.7 九九乘法表输出。工整打印输出常用的九九乘法表，格式不限。

```
1 for i in range(1,10):
2     for j in range(1,i+1):
3         print("{}*{}={:2} ".format(j,i,i*j), end='')
4     print('')
```

3 Python 编程之实例解析

3.1 Python 语法元素分析

程序元素：注释、缩进、变量、常量、表达式、输入、输出、分支、循环
回顾温度转换程序：

```
1  # TempConvert.py
2  val = input(" 请输入带温度表示符号的温度值（例如：32C）：")
3  if val[-1] in ['C', 'c']:
4      f = 1.8 * float(val[0:-1]) + 32
5      print(" 转换后的温度为：%.2fF"%f)
6  elif val[-1] in ['F', 'f']:
7      c = (float(val[0:-1]) - 32) / 1.8
8      print(" 转换后的温度为：%.2fC"%c)
9  else:
10     print(" 输入有误")
```

3.1.1 缩进

缩进用以在 Python 中标明代码的层次关系，是 Python 语言中表明程序框架的唯一手段。1 个缩进 = 4 个空格。

```
if val[-1] in ['C', 'c']:
    f = 1.8 * float(val[0:-1]) + 32
    print(" 转换后的温度为：%.2fF"%f)
elif val[-1] in ['F', 'f']:
    c = (float(val[0:-1]) - 32) / 1.8
    print(" 转换后的温度为：%.2fC"%c)
else:
    print(" 输入有误")
```

3.1.2 注释

注释是程序员在代码中加入的说明信息，不被计算机执行。注释的两种方法：

- 单行注释以 # 开头

```
#Here are the comments
```

- 多行注释以''' 开头和结尾

```
'''
if val[-1] in ['C','c']:
    f = 1.8 * float(val[0:-1]) + 32
    print(" 转换后的温度为: %.2fF"%f)
else:
    print(" 输入有误")'''
```

Spyder 常用快捷键: Ctrl + 1: 注释/反注释; Ctrl + 4/5: 块注释/块反注释

```
#=====
# # Ctrl + 4 块注释示例
# val = input(" 请输入带温度表示符号的温度值 (例如: 32C): ")
# if val[-1] in ['C','c']:
#     f = 1.8 * float(val[0:-1]) + 32
#     print(" 转换后的温度为: %.2fF"%f)
#=====
```

3.1.3 常量与变量

常量是程序中值不发生改变的元素。

- ◆ 好处: 例如, 程序中含有一个常量, $\pi=3.14$, 如果程序中多次使用 π , 当我们需要更精确的值时, 直接修改常量定义, 而不需要每一处使用都修改具体值。

变量是程序中值发生改变或者可以发生改变的元素。

- ◆ 在 Python 语言中, 变量和常量使用上基本没有区别。

3.1.4 命名

命名是给程序元素关联一个标识符, 保证唯一性。变量和常量都需要一个名字。

命名规则

- 大小写字母、数字和下划线的组合，但首字母只能是大小写字母或下划线，不能使用空格
- 中文等非字母符号也可以作为名字

以下是合法命名的标识符:python_is_good、python_is_not_good、_is_it_a_question_、python 语言

常量、变量与命名

- 标识符对大小写敏感，不能与保留字相同

- Python 3.x 保留字列表 (33 个)

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

3.1.5 表达式

表达式是程序中产生或计算新数据值的一行代码。

Python 语言的 33 个保留字或者操作符可以产生符合语法的表达式。例如

```
# 将字符串 '28C' 赋给变量 val
val = '28C'
```

- 在使用变量前必须对其赋值，否则编译器报错

表达式中空格的使用

- 不改变缩进相关的空格数量
- 空格不能将命名分割
- 增加空格增加程序可读性

如果 `val = "28C"`，则 `val[-1]` 是最后一个字符串 "C"；前两个字符组成的子串可以用 `val[0:2]` 表示，它表示一个从 `[0,2)` 的区间。

由于约定用户输入的最后一个字符是 C 或者 F，之前是数字，所以通过 `val[0:-1]` 来获取除最后一个字符外的字符串

3.1.6 输入函数

Input() 函数从控制台获得用户输入。其语法如下：

< 变量 >= input(< 提示性文字 >)

获得的用户输入以字符串形式保存在 < 变量 > 中。

```
>>> val = input(" 请输入带温度表示符号的温度值 (例如: 32C): ")
请输入带温度表示符号的温度值 (例如: 32C):35C
>>>
```

3.1.7 分支语句

分支语句控制程序运行，根据判断条件选择程序执行路径。其语法如下：

```
if < 条件 1 成立 >:
    < 表达式组 1>
elif < 条件 2 成立 >:
    < 表达式组 2>
    .....
elif < 条件 N-1 成立 >:
    < 表达式组 N-1>
else:
    < 表达式组 N>
```

3.1.8 赋值语句

赋值语句使用等号给变量赋值。

```
f=1.8*float(str[0:-1]) + 32
```

同步赋值语句：同时给多个变量赋值（先运算右侧 N 个表达式，然后同时将表达式结果赋给左侧）。

```
f=1.8*float(str[0:-1]) + 32
< 变量 1>, ..., < 变量 N > = < 表达式 1 >, ..., < 表达式 N >
```

例：将变量 x 和 y 交换，采用单个赋值，需要 3 行语句：即通过一个临时变量 t 缓存 x 的原始值，然后将 y 值赋给 x ，再将 x 的原始值通过 t 赋值给 y 。采用同步赋值语句，仅需要一行代码。

单个赋值与同步赋值——变量 x 和 y 交换

```
>>> t=x
>>> x=y
>>> y=t
```

```
>>> x,y=y,x
```

3.1.9 输出函数

`print()` 函数用来输出字符信息，或以字符形式输出变量。

`print()` 函数可以输出各种类型变量的值。

`print()` 函数通过 `%` 来选择要输出的变量。

例 3.1 用户输入两个数字，计算它们的平均数，并输出平均数。

```
num1=input('The first number is ')
num2=input('The second number is ')
avg_num=(float(num1)+float(num2))/2
print('The average number is %f' % avg_num)
```

3.1.10 循环语句

循环语句控制程序运行，根据判断条件或计数条件确定一段程序的运行次数。

计数循环，基本过程如下

```
for i in range (< 计数值 >):
    < 表达式 1>
```

例 3.2 例如，使某一段程序连续运行 10 次，循环代码如下：

```
1 # 变量 i 用于计数
2 for i in range (10):
3     <source code>
```


3.2 程序编写模板

3.2.1 input-print 模板

input-print 模板

- 用户输入: `input()` 获得输入
- 运算部分: 根据算法实现
- 结果输出: `print()` 输出结果

3.2.2 initial-print 模板

initial-print 模板

- 初始变量: 运算需要的初始值
- 运算部分: 根据算法实现
- 结果输出: `print()` 输出结果

任何输入输出类型的组合都可以看成“模板”，例如：`input` 输入 - 文件 `write` 输出

3.3 turtle 库与蟒蛇绘制程序



图 3-1 Python 蟒蛇绘制实例

Python 英文是蟒蛇的意思，通过下面的例子，来实践用 Python 语言输出如图 3-1 的图形效果。

```
1 import turtle                                # 引入 turtle 的函数库
2 def drawSnake(rad,angle,len,neckrad):        # 定义绘制蟒蛇功能函数
3     for i in range(len):
4         turtle.circle(rad,angle)             # 让小乌龟沿着一个圆形爬行，参数
        ↳ rad 描述圆形轨迹半径的位置。如果 rad 为正值，这个半径在小
        ↳ 乌龟运行的左侧 rad 远位置处；如果 rad 为负值，则半径在小
        ↳ 乌龟运行的右侧。参数 angle 表示小乌龟沿着圆形爬行的弧度值
5         turtle.circle(-rad,angle)
6     turtle.circle(rad,angle/2)
7     turtle.fd(rad)                            # 表示小乌龟向前直线爬行移动，其参数表
        ↳ 示爬行的距离，也可以用 turtle.forward() 表示乌龟向前直线爬行
        ↳ 移动
8     turtle.circle(neckrad+1,180)
9     turtle.fd(rad*2/3)
10
11 def main():
12     turtle.setup(1300,800,0,0)               #turtle.setup(width, height,
        ↳ startx, starty) 用于启动一个图形窗口，四个参数分别表示启动窗
        ↳ 口的宽度，高度，窗口左上角在屏幕中的坐标位置
13     pythonsize=30
14     turtle.pensize(pythonsize)               #turtle.pensize() 函数表示小乌龟运动
        ↳ 轨迹的宽度，它包含一个输入参数，这里我们把它设为 30 像素，用
        ↳ pythonsize 变量表示
15     turtle.pencolor("blue")                  #turtle.pencolor() 函数表示小乌龟运
        ↳ 动轨迹的颜色，采用 RGB，同 turtle.pencolor(“#3B9909”)
16     turtle.seth(-40)                         #turtle.seth(angle) 函数表示小乌龟启
        ↳ 动时运动的方向，输入参数为角度值。0 表示向东，90 度向北，180
        ↳ 度向西，270 度向南；负值表示相反方向。-40 度即向东南方向 40 度
17     drawSnake(40,80,5,pythonsize/2)         # 调用绘制蟒蛇功能函数
18
19 main()
```

观察上述代码，不难发现，首先：这个代码没有 input 输入也没有 print 输出；其次：代码绝大部分都是 <a>.() 类型的函数运行，仅有 1 个赋值表达式；最后，代码通过 def 分割成了若干块。

3.3.1 蟒蛇绘制程序分析

```
import turtle
```

`import` 是一个关键字，用来引入一些外部库，这里的含义是引入一个名字叫 `turtle` 的函数库。`turtle` 函数库是 Python 的内置图形化模块，Python 3 系列版本安装目录的 `Lib` 文件夹下可以找到 `turtle.py` 文件。`turtle` 库绘制图形的命令简单且直观。

更多信息请查阅[turtle 库官方文档](#)。

```
def drawSnake(rad,angle,len,neckrad):  
.....  
def main():
```

def 定义函数

- `def` 用于定义函数，这段程序中，共出现两次 `def` 关键词，包含两个函数 `drawSnake` 和 `main`。
- 函数是一组代码的集合，用于表达一个功能，或者说，函数表示一组代码的归属，函数名称是这段代码的名字。
- `def` 所定义的函数在程序中未经调用不能直接执行，需要通过函数名调用才能够执行。

可以看到，两个 `def` 语句定义的函数所包含语句与 `def` 行存在缩进关系，`def` 后连续的缩进语句都是这个函数的一部分。

由于 `def` 定义的函数在程序中未经调用不会被执行，整个程序第一条执行的语句是 `main()`，它表示执行名字为 `main()` 的函数。

从而，该程序跳转到 `main()` 函数定义的一组语句中执行，即开始执行 `turtle.setup()` 语句。`main()` 函数给出了小乌龟爬行的窗体大小，爬行轨迹颜色和宽度以及初始爬行的方位。

同样的，`main()` 函数的最后一条语句调用了 `drawSnake()` 函数，当执行到这条语句时，程序跳转到 `drawSnake()` 函数中运行启动绘制蟒蛇功能。`drawSnake` 函数有四个参数，根据调用时给出的参数，分别将 40 传递给 `rad`、80 给 `angle`，5 给 `len`，15 给 `neckrad`。

3.3.2 Turtle 库快速参考

Python Quick Reference Series

Python快速参考

turtle库

引入方式

```
>>>import turtle
>>>from turtle import *
```

控制画笔绘制状态的函数

```
pendown()      | pd()      | down()
penup()        | pu()      | up()
pensize(wid )  | width(wid)
```

控制画笔颜色和字体函数

```
color()          reset()
begin_fill()     end_fill()
filling()        clear()
screensize()
showturtle()    | st()
hideturtle()   | ht()
isvisible()
write(arg,move=False,align="left"
,font=("Arial",8,"normal") )
```

控制画笔运动的函数

```
forward(distance) | fd(distance)
backward(distance)| bk(distance)
|back(distance)
right(angle)      | rt(angle)
left(angle)       | lt(angle)
setheading(to_angle)
position()        | pos()
goto(x,y )
setposition(x,y ) | setpos(x,y )
circle(radius,extent ,steps )
dot(size ,*color) radians()
stamp()           speed(speed )
clearstamp(stamp_id)
clearstamps(n )   undo()
speed(speed )     heading()
towards(x,y )     distance(x,y )
xcor()            ycor()
setx(x)           sety(y)
home()            undo()
degrees(fullcircle = 360.0)
```

TurtleScreen/Screen类的函数

```
bgcolor(*args)
bgpic(picname )
clearscreen()
resetscreen()
screensize(cwid ,canvh,bg )
tracer(n ,delay )
listen(xdummy ,ydummy )
onkey( (fun,key)
onkeyrelease( (fun,key)
onkeypress(fun,key )
onscreenclick(fun,btn=1,add )

getcanvas()
getshapes()
turtles()
window_height()
window_width()
bye()
exitonclick()
title(titlestring)
setup(wid=_CFG["wid"],h=_CFG["h"],
startx=_CFG["leftright"],
starty=_CFG["topbottom"])
```

3.4 函数库的引用

Python 语言的魅力在于大量使用外部函数库

- 包含在安装包中的函数库：math, random, turtle 等
- 其他函数库：用户根据需求安装

Python 对函数库引用的两种方式

第一种方式：

```
import < 库名 >
```

例如：import turtle

如果需要用到函数库中函数，需要使用：

```
< 库名 >.< 函数名 >
```

例

```
>>> import turtle
>>> turtle.fd(100)
```

第二种方式：

```
from < 库名 > import < 函数名 >
from < 库名 > import *
```

调用函数不需要 < 库名 >，直接使用 < 函数名 > 例

```
>>> from turtle import < 函数名 >
>>> fd(100)
```

两种引用方式的区别

这两种引用方式对程序运行没有区别，需要注意：如果采用第一种方式，用户自定义的函数名字可以和库中函数的名字一样，例如，程序中可以定义自己的 fd() 函数；如果采用第二种方式，用户程序中不能用函数库中的名字定义函数，例如：程序不能定义新的 fd() 函数，因为库 turtle 中的 fd() 函数也是直接通过 fd() 调用。

4 Python 编程之数据类型

数据从不同角度有不同的含义，这样一个数据：10,011,101，该怎样解释呢？它可以是：

- 1 个二进制数字或者 1 个十进制数字
- 一段文本或者用 “,” 分割的 3 个数字

程序设计语言不允许存在语法歧义，因此，需要明确说明数据的含义，这就是“类型”的作用。

4.1 Python 中类型的概念

类型是编程语言对数据的一种划分，本课程主要介绍 6 种 Python 语言中的类型：

- 数字类型、字符串类型
- 元组类型、列表类型
- 文件类型、字典类型（后续章节介绍）

4.1.1 数字类型

程序元素：010/10，存在多种可能：

- 表示十进制整数值 10
- 类似人名一样的字符串

数字类型对 Python 语言中数字的表示和使用进行了定义和规范。

Python 语言包括三种数字类型：整数类型、浮点数类型、复数类型。

整数类型

与数学中的整数概念一致，没有取值范围限制。

例 4.1 pow(x, y) 函数：计算 x^y 。

```
Code1: >>> pow(2,10) , pow(2,15)
```

```
Code2: >>> pow(2, 1000)
```

```
Code3: >>> pow(2, pow(2,15))
```

例 4.2

- 1010, 99, -217
- 0x9a, -0X89 (0x, 0X 开头表示 16 进制数)
- 0b010, -0B101 (0b, 0B 开头表示 2 进制数)
- 0o123, -0O456 (0o, 0O 开头表示 8 进制数)

浮点数类型

带有小数点及小数的数字。Python 语言中浮点数的数值范围存在限制，小数精度也存在限制，这种限制与在不同计算机系统有关。

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
    ↳ max_10_exp=308, min=2.2250738585072014e-308,
    ↳ min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
    ↳ epsilon=2.220446049250313e-16, radix=2, rounds=1)
>>>
```

例 4.3

- 0.0, -77., -2.17
- 96e4, 4.3e-3, 9.6E5 （科学计数法）

科学计数法使用字母“e”或者“E”作为幂的符号，以 10 为基数。科学计数法含义如下： $\langle a \rangle e \langle b \rangle = a \times 10^b$

复数类型

与数学中的复数概念一致， $z = a + bj$ ， a 是实数部分， b 是虚数部分， a 和 b 都是浮点类型，虚数部分用 j 或者 J 标识。

例 4.4

```
>>> 12.3+4j
(12.3+4j)
>>> -5.6+7j
(-5.6+7j)
```

例 4.5 对于复数 $z = 1.23e-4+5.6e+89j$ ，Python 可以用 `z.real` 获得实数部分，`z.imag` 获得虚数部分：

```
>>> z=1.23e-4+5.6e+89j
>>> z.real,z.imag
(0.000123, 5.6e+89)
>>>
```

数字类型的关系 三种类型存在一种逐渐“扩展”的关系：

- 整数 -> 浮点数 -> 复数 (整数是浮点数特例, 浮点数是复数特例)
 - 不同数字类型之间可以进行混合运算, 运算后生成结果为最宽类型
- 例 4.6 $123 + 4.0 = 127.0$ (整数 + 浮点数 = 浮点数)

三种数字类型可以相互转换, 函数: `int()`, `float()`, `complex()`。

```
>>> int(4.9)
4          # (直接去掉小数部分)
>>> float(5)
5.0       # (增加小数部分)
>>> complex(9)
(9+0j)
```

数字类型的判断, 函数: `type(x)`, 返回 `x` 的类型, 适用于所有类型的判断。

```
>>> type(4.5)
<class 'float'>
>>> type(5.2+7j)
<class 'complex'>
>>> type(3)
<class 'int'>
```

常见的数字类型运算操作如表 4-1 所示。

表 4-1 数字类型的运算

运算符和运算函数	操作含义
<code>x+y</code>	<code>x</code> 与 <code>y</code> 之和
<code>x-y</code>	<code>x</code> 与 <code>y</code> 之差
<code>x*y</code>	<code>x</code> 与 <code>y</code> 之积
<code>x/y</code>	<code>x</code> 与 <code>y</code> 之商
<code>x//y</code>	不大于 <code>x</code> 与 <code>y</code> 之商的最大整数
<code>x%y</code>	<code>x</code> 与 <code>y</code> 之商的余数
<code>+x</code>	<code>x</code>
<code>-x</code>	<code>x</code> 的负值
<code>x**y</code>	<code>x</code> 的 <code>y</code> 次幂
<code>abs(x)</code>	<code>x</code> 的绝对值
<code>divmod(x,y)</code>	<code>(x//y,x%y)</code>
<code>pow(x,y)</code>	<code>x</code> 的 <code>y</code> 次幂

4.1.2 字符串类型

字符串是用双引号"" 或者单引号' 括起来的一个或多个字符。字符串可以保存在变量中，也可以单独存在。

可以用 `type()` 函数测试一个字符串的类型，如：

```
>>> string="HappyPython"
>>> type(string)
<class 'str'>
```

Python 语言转义符 “\” 输出带有引号的字符串。使用 “\\” 输出带有转义符的字符串。

```
>>> print("\"HappyPython\"")
"HappyPython"
>>> print("\\Happy\\Python")
\\Happy\\Python
```

字符串是一个字符序列，字符串最左端位置标记为 0，依次增加。字符串中的编号叫做“索引”。Python 中字符串索引从 0 开始，一个长度为 L 的字符串最后一个字符的位置是 L-1。

H	a	p	p	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

单个索引辅助访问字符串中的特定位置，格式为

```
<string>[< 索引 >]
```

Python 同时允许使用负数从字符串右边末尾向左边进行反向索引，最右侧索引值是-1。

```
>>> string="Happy Python"
>>> print(string[6])
P
>>> print(string[-6])
P
```

可以通过两个索引值确定一个位置范围，返回这个范围的子串，格式：

```
| <string>[<start>:<end>]
```

其中, start 和 end 都是整数型数值, 这个子序列从索引 start 开始直到索引 end 结束, 但不包括 end 位置。

```
| >>> print(string[0:6])  
| Happy
```

字符串之间可以通过 + 或 * 进行连接, 加法操作 (+) 将两个字符串连接成一个新的字符串; 乘法操作 (*) 生成一个由其本身字符串重复连接而成的字符串。

```
| >>> "Happy"+"Python"  
| 'HappyPython'  
| >>> 3*"Love"  
| 'LoveLoveLove'
```

大多数数据类型都可以通过 str() 函数转换为字符串。len() 函数能返回一个字符串的长度。

```
| >>> calculation=pow(2,100)  
| >>> len(str(calculation))  
| 31
```

例 4.7 字符串使用实例, 输入一个月份数字, 返回对应月份名称缩写。

这个问题的 IPO 模式是:

- 输入: 输入一个表示月份的数字 (1-12)
- 处理: 利用字符串基本操作实现该功能
- 输出: 输入数字对应月份名称的缩写

具体实现代码如下:

```
1 # month.py  
2 months="JanFebMarAprMayJunJulAugSepOctNovDec" # 将所有月份名称缩写  
   ↳ 存储在字符串中  
3 n=input(" 请输入月份数 (1-12):")  
4 pos=(int(n)-1) * 3  
5 monthAbbrev=months[pos:pos+3] # 在字符串中截取子串来查找特定月份  
6 print(" 月份简写是"+monthAbbrev+".")
```

运行效果如下：

```
请输入月份数 (1-12):5
月份简写是 May.
```

字符串的操作有如表 4-2 处理方法：

表 4-2 字符串处理方法

运算符和运算函数	操作含义
+	连接
*	重复
<string>[]	索引
<string>[:]	剪切
len(<string>)	长度
<string>.upper()	字符串中字母大写
<string>.lower()	字符串中字母小写
<string>.strip()	去两边空格及去指定字符
<string>.split()	按指定字符分割字符串为数组
<string>.join()	连接两个字符串序列
<string>.find()	搜索指定字符串
<string>.replace()	字符串替换
for <var> in <string>	字符串迭代

可以通过 for 和 in 组成的循环来遍历字符串中每个字符，格式如下：

```
for <var> in <string>:
    操作
```

用转义符可以在字符串中表达一些不可直接打印的信息，例如：用 \n 表示换行。

```
>>> print("Hello\nWorld\n\nGoodbye 32\n")
Hello
World

Goodbye 32
```

字符串类型格式化采用 format() 方法，基本使用格式是：

| < 模板字符串 >.format(< 逗号分隔的参数 >)

其中，< 模板字符串 > 由一系列的槽组成，用来控制修改字符串中嵌入值出现的位置，其基本思想是将 format() 方法的 < 逗号分隔的参数 > 中的参数按照序号关系替换到 < 模板字符串 > 的槽中。

槽用大括号 {} 表示，如果大括号中没有序号，则按照出现顺序替换，如图 4-1 所示。

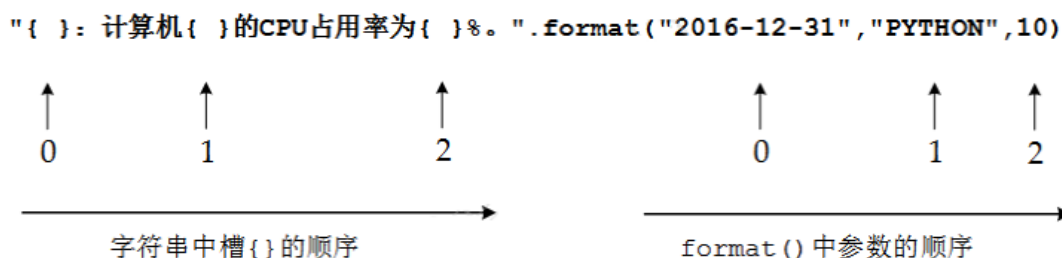


图 4-1 format() 方法的槽顺序和参数顺序

如果大括号中指定了使用参数的序号，按照序号对应参数替换，如图 4-2 所示。调用 format() 方法后会返回一个新的字符串，参数从 0 开始编号。

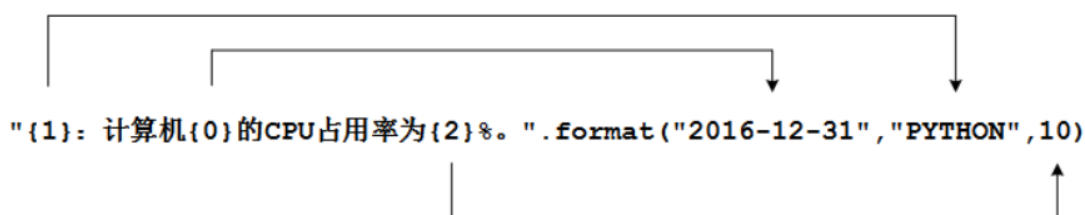


图 4-2 format() 方法槽与参数的对应关系

例如：

```
>>> "{ } : 计算机 { } 的 的 CPU 占用率为 { }% 。"
↪ ".format("2016-12-31", "PYTHON", 10)
'2016-12-31 : 计算机 PYTHON 的 的 CPU 占用率为 10% 。'
```

format() 方法可以非常方便地连接不同类型的变量或内容，如果需要输出大括号，采用表示，表示，例如：

```
>>> "{}{}{}".format(" 圆周率是", 3.1415926, "...")
' 圆周率是 3.1415926...'
>>> " 圆周率 {{{1}}{2}} 是 {0}".format(" 无理
↪ 数", 3.1415926, "...")
' 圆周率 {3.1415926...} 是 无理数'
```

```
>>> s=" 圆周率 {{{1}{2}}} 是 {0}" # 大括号本身是字符串的一部分
>>> s
' 圆周率 {{{1}{2}}} 是 {0}'
>>> s.format(" 无理数",3.1415926,"...") # 当调用 format() 时解析
↳ 大括号
' 圆周率 {3.1415926...} 是 无理数'
```

format() 方法中 < 模板字符串 > 的槽除了包括参数序号，还可以包括格式控制信息。此时，槽的内部样式如下：

{< 参数序号 > : < 格式控制标记 > }

其中，< 格式控制标记 > 用来控制参数显示时的格式，如图 4-3 所示。

:	<填充>	<对齐>	<宽度>	,	<.精度>	<类型>
	用于填充的 单个字符	< 左对齐 > 右对齐 ^ 居中对齐	槽的设定输出 宽度	数字的千位 分隔符 适用于整数 和浮点数	浮点数小数 部分的精度 或 字符串的最 大输出长度	整数类型 b, c, d, o, x, X 浮点数类型 e, E, f, %

图 4-3 槽中格式控制标记的字段

< 格式控制标记 > 包括：< 填充 >< 对齐 >< 宽度 >,<. 精度 >< 类型 >6 个字段，这些字段都是可选的，可以组合使用，逐一介绍如下。

< 填充 >、< 对齐 > 和 < 宽度 > 是 3 个相关字段。< 宽度 > 指当前槽的设定输出字符宽度，如果该槽对应的 format() 参数长度比 < 宽度 > 设定值大，则使用参数实际长度。如果该值的实际位数小于指定宽度，则位数将被默认以空格字符补充。< 对齐 > 指参数在 < 宽度 > 内输出时的对齐方式，分别使用 <、> 和 ^ 三个符号表示左对齐、右对齐和居中对齐。< 填充 > 指 < 宽度 > 内除了参数外的字符采用什么方式表示，默认采用空格，可以通过 < 填充 > 更换。例如：

```
>>> s = "PYTHON"
>>> str1="{0:30}".format(s)
>>> str1
'PYTHON'
>>> len(str1)
30
>>> "{0:>30}".format(s)
```

```
'
                                PYTHON'
>>> "{0: ^30}".format(s)
'*****PYTHON*****'
>>> "{0: -^30}".format(s)
'-----PYTHON-----'
>>> "{0: 3}".format(s)
'PYTHON'
```

< 格式控制标记 > 中逗号 “,” 用于显示数字的千位分隔符，例如：

```
>>> "{0: -^20,}".format(1234567890)
'---1,234,567,890---'
>>> "{0: -^20}".format(1234567890) # 对比输出
'-----1234567890-----'
>>> "{0: -^20,}".format(12345.67890)
'----12,345.6789-----'
```

<. 精度 > 表示两个含义，由小数点 “.” 开头。对于浮点数，精度表示小数部分输出的有效位数。对于字符串，精度表示输出的最大长度。例如：

```
>>> "{0: ,.2f}".format(12345.67890)
'12,345.68'
>>> "{0: H^20.3f}".format(12345.67890)
'HHHHH12345.679HHHHH'
>>> "{0: .4}".format("PYTHON")
'PYTH'
```

< 类型 > 表示输出整数和浮点数类型的格式规则。对于整数类型，输出格式包括 6 种：

- b: 输出整数的二进制方式；
- c: 输出整数对应的 Unicode 字符；
- d: 输出整数的十进制方式；
- o: 输出整数的八进制方式；
- x: 输出整数的小写十六进制方式；
- X: 输出整数的大写十六进制方式；

例如：

```
>>> "{0:b},{0:c},{0:d},{0:o},{0:x},{0:X}".format(425)
'110101001,,425,651,1a9,1A9'
```

对于浮点数类型，输出格式包括 4 种：

- e: 输出浮点数对应的小写字母 e 的指数形式；
- E: 输出浮点数对应的大写字母 E 的指数形式；
- f: 输出浮点数的标准浮点形式；
- %: 输出浮点数的百分形式

浮点数输出时尽量使用 <.精度> 表示小数部分的宽度，有助于更好控制输出格式。

```
>>> "{0:e},{0:E},{0:f},{0:%}".format(3.14)
'3.140000e+00,3.140000E+00,3.140000,314.000000%'
>>> "{0:.2e},{0:.2E},{0:.2f},{0:.2%}".format(3.14)
'3.14e+00,3.14E+00,3.14,314.00%'
```

4.1.3 元组类型

定义 4.1 (元组) 元组 (tuple) 是包含多个元素的类型，元素之间用逗号分割。例如：t1 = 123,456, “hello”。

元组可以是空的，如 t2=(), 也可只包含一个元素，如 t3=123。元组外侧可以使用括号，也可以不使用。

元组的特点：

- 元组中元素可以是不同类型
- 一个元组也可以作为另一个元组的元素，此时，作为元素的元组需要增加括号，从而避免歧义

```
>>> tuple1=123,456,("Happy","Python")
```

- 元组中各元素存在先后关系，可以通过索引访问元组中元素

```
>>> tuple[1]
456
```

- 元组定义后不能更改，也不能删除

```
>>> tuple[1]=666
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
```

```
tuple[1]=666
TypeError: 'tuple' object does not support item assignment
```

- 与字符串类型类似，可以通过索引区间来访问元组中部分元素

```
tuple[1:]
(456, ('Happy', 'Python'))
```

- 与字符串一样，元组之间可以使用 + 号和 * 号进行运算

元组总结

- (1) Python 语言中的元组类型定义后不能修改。
- (2) 不可变的 tuple 有什么意义呢？因为 tuple 不可变，所以代码更安全
- (3) 与字符串类型类似，可对元组中的元素进行索引、元组之间可以使用 + 号和 * 号进行运算
- (4) 如果仅考虑代码的灵活性，也可以用列表类型代替元组类型

4.1.4 列表类型

定义 4.2 (列表) 列表 (list) 是有序的元素集合，其元素可通过索引访问单个元素。

列表与元组的异同

列表与元组类似

- (1) 列表中每个元素类型可以不一样
- (2) 访问列表中元素时采用索引形式

列表与元组不同，列表大小没有限制，可以随时修改

针对列表有一些基本操作，如表 4-3 所示，这些操作与字符串操作类似。

表 4-3 列表的操作

列表操作符	操作符含义
< list1 > + < list2 >	连接两个列表
< list > * < 整数类型 >	对列表进行整数次重复
< list > [< 整数类型 >]	索引列表中的元素
len(< seq >)	列表中元素个数
< list > [< 整数类型 > : < 整数类型 >]	取列表的一个子序列

续下页

续表 4-3 列表的操作

列表操作符	操作符含义
for < var > in < list >	对列表进行循环列举
< expr > in < list >	成员检查，判断 <expr> 是否在列表中

列表操作示例

```
>>> vlist=[0,1,2,3,4]
>>> vlist*2
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
>>> len(vlist[2:])
3
>>> for i in vlist[3:]:
    print(i)

3
4
>>> 3 in vlist
True
```

列表相关方法如表 4-4 所示。

表 4-4 列表相关方法

列表方法	方法含义
< list > . append (x)	将元素 x 增加到列表的最后
< list > . sort ()	将列表元素排序
< list > . reverse ()	将序列元素反转
< list > . index ()	返回第一次出现元素 x 的索引值
< list > . insert (i, x)	在位置 i 处插入新元素 x
< list > . count (x)	返回元素 x 在列表中的数量
< list > . remove (x)	删除列表中第一次出现的元素 x
< list > . pop (i)	取出列表中位置 i 的元素，并删除它

列表方法示例

```
>>> vlist.insert(2,10)
>>> vlist
[0, 1, 10, 2, 3, 4]
>>> vlist=[0,1,2,3,4]
>>> vlist.append("python")
>>> vlist
[0, 1, 2, 3, 4, 'python']
>>> vlist.reverse()
>>> vlist
['python', 4, 3, 2, 1, 0]
>>> vlist.pop(2)
3
>>> vlist
['python', 4, 2, 1, 0]
```

♠ 对于字符串，可以通过 `split()` 函数，将字符串拆分成一个列表，默认以空格分割。

```
>>> "Pythhon is a excellent language".split()
['Pythhon', 'is', 'a', 'excellent', 'language']
```

4.2 math 库和 random 库

4.2.1 math 库

math 库中常用的数学函数如表 4-5 所示。

表 4-5 math 库中常用的数学函数

函数	数学表示	含义
圆周率 π	π	π 的近似值，15 位小数
自然常数 e	e	e 的近似值，15 位小数
<code>ceil(x)</code>	$\lceil x \rceil$	对浮点数向上取整
<code>floor(x)</code>	$\lfloor x \rfloor$	对浮点数向下取整
<code>pow(x,y)</code>	x^y	计算 x 的 y 次方
<code>log(x)</code>	$\lg x$	以 e 为基的对数

续下页

续表 4-5 math 库中常用的数学函数

函数	数学表示	含义
$\log_{10}(x)$	$\log_{10}x$	以 10 为基的对数
$\text{sqrt}(x)$	\sqrt{x}	平方根
$\text{exp}(x)$		e 的 x 次幂
$\text{degrees}(x)$		将弧度值转换成角度
$\text{radians}(x)$		将角度值转换成弧度
$\sin(x)$	$\sin x$	正弦函数
$\cos(x)$	$\cos x$	余弦函数
$\tan(x)$	$\tan x$	正切函数
$\text{asin}(x)$	$\arcsin x$	反正弦函数, $x \in [-1.0, 1.0]$
$\text{acos}(x)$	$\arccos x$	反余弦函数, $x \in [-1.0, 1.0]$

4.2.2 random 库

random 库中常用的数学函数如表 4-6 所示。

表 4-6 random 库中常用的数学函数

函数	含义
$\text{seed}(x)$	给随机数一个种子值, 默认随机种子是系统时钟
$\text{random}()$	生成一个 $[0, 1.0)$ 之间的随机小数
$\text{uniform}(a,b)$	生成一个 a 到 b 之间的随机小数
$\text{randint}(a,b)$	生成一个 a 到 b 之间的随机整数
$\text{randrange}(a,b,c)$	随机生成一个从 a 开始到 b 以 c 递增的数
$\text{choice}(<\text{list}>)$	从列表中随机返回一个元素
$\text{shuffle}(<\text{list}>)$	将列表中元素随机打乱
$\text{sample}(<\text{list}>,k)$	从指定列表随机获取 k 个元素

随机数库使用示例:

```
>>> from random import * # 导入随机数库
>>> random()
0.4941953366643348
>>> uniform(1,10)
3.89818562842704
>>> randint(1,10)
```

```
4
>>> randrange(0,10,2)
0
>>> randrange(0,10,4)
8
>>> list1=[0,1,2,3,4,5,6,7,8,9]
>>> choice(list1)
1
>>> shuffle(list1)
>>> list1
[0, 9, 4, 6, 5, 2, 3, 8, 7, 1]
>>> sample(list1,5)
[5, 3, 8, 4, 6]
```

调用 `seed()` 函数，重置随机种子：

```
>>> seed(10)
>>> uniform(1,10)
6.142623352209221
>>> uniform(1,10)
4.860001492076032
```

再次设定相同的随机种子：

```
>>> seed(10)
>>> uniform(1,10)
6.142623352209221
>>> uniform(1,10)
4.860001492076032
```

◆ 当设定相同的种子后，每次调用随机函数后生成的随机数都是相同的，这就是随机种子的作用。

◆ 因为计算机是一个确定设备，不能生成真正的随机数。所以，由计算机产生的随机数都是由一个种子开始的伪随机序列。

◆ 相同的随机种子产生相同的伪随机数序列，也有利于程序的验证执行。

4.2.3 π 的计算

例 4.8 圆周率 π 是一个无理数，没有任何一个精确公式能够计算 π 值，的计算只能采用近似算法，国际公认的 π 值计算采用蒙特卡洛方法。

蒙特卡洛 (Monte Carlo)

蒙特卡洛 (Monte Carlo) 方法，又称随机抽样或统计试验方法。当所求解问题是某种事件出现的概率，或某随机变量期望值时，可以通过某种“试验”的方法求解。

简单说，蒙特卡洛是利用随机试验求解问题的方法

应用蒙特卡洛方法求解圆周率 π 的步骤是这样的：

1 首先构造一个单位正方形和 $1/4$ 圆，如图 4-4

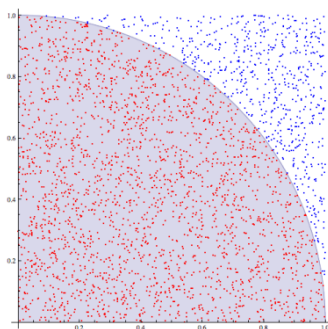


图 4-4 单位正方形和 $1/4$ 圆

2 随机向单位正方形内抛洒大量点，对于每个点，可能在圆内或者圆外，当随机抛点数量达到一定程度，圆内点将构成圆的面积，全部抛点将构成矩形面积。圆内点数除以圆外点数就是面积之比，即 $\pi/4$ 。随机点数量越大，得到的 π 值越精确。

π 计算问题的 IPO 表示如下：

- 输入：抛点的数量
- 处理：对于每个抛洒点，计算点到圆心的距离，通过距离判断该点在圆内或是圆外，统计在圆内点的数量
- 输出： π 值

π 计算问题的 python 实现:

```
1  # pi.py
2  from random import random # 导入 random 库
3  from math import sqrt     # 导入 math 库中的平方根函数
4  from time import clock    # 导入 time 库中的计时函数
5  DARTS = 1200              # 设定 DARTS 点数量
6  hits = 0
7  clock()
8  for i in range(1,DARTS):
9      x, y = random(), random() #random() 函数给出随机坐标 (x,y)
10     dist = sqrt(x**2 + y**2)  #sqrt() 计算抛点到原点距离
11     if dist <= 1.0:           # 判断该点是否在圆内
12         hits = hits + 1      # 圆内点计数
13 pi = 4 * (hits/DARTS)        # 计算比值
14 print("Pi 的值是 %s" % pi)
15 print(" 程序运行时间是 %-5.5ss" % clock())
```

运行结果:

```
>>>
```

```
Pi 的值是 3.15
```

```
程序运行时间是 0.001s
```

代码主体是一个循环，模拟抛洒多个点的过程。对于一个抛点，通过 `random()` 函数给出随机的坐标值 (x,y) ，首先利用开方函数 `sqrt()` 计算抛点到原点距离；然后通过 `if` 语句判断这个距离是否落在圆内；最终，根据总抛点落入圆内的数量，计算比值，从而得到 π 值。

由于 DARTS 点数量较少，的值不是很精确，进一步增加 DARTS 数量，能够进一步提高 π 的计算精度。DARTS 在 2^{30} 数量级上， π 的值就相对准确了。

蒙特卡洛方法提供了一个利用计算机中随机数和随机试验解决现实中无法通过公式求解问题的思路。它广泛应用在金融工程学，宏观经济学，计算物理学（如粒子输运计算、量子热力学计算、空气动力学计算）等领域中。

5 Python 编程之控制结构

5.1 程序基本结构

任何算法 (程序) 都可以由顺序结构、选择结构和循环结构这三种基本结构组合来实现。

5.1.1 顺序结构

顺序结构中，按语句的自然顺序依次执行。用顺序结构描述将华氏温度 F 转换成摄氏温度 C 的流程如图 5-1 所示。

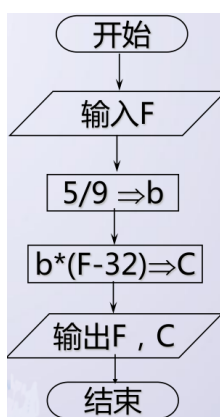


图 5-1 顺序结构描述华氏温度与摄氏温度转换

5.1.2 选择结构

选择结构也叫分支结构，是指在算法中通过对条件的判断，根据条件是否成立而选择不同流向的算法结构。有如下图 ?? 两种结构。

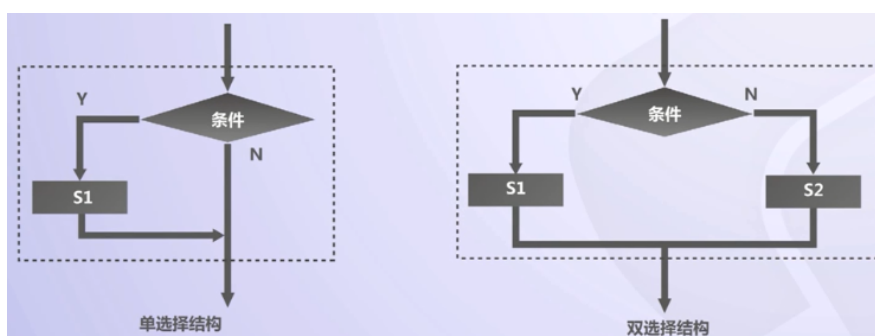


图 5-2 单选择结构与双选择结构

单选择结构与双选择结构的区别在于，前者在不满足条件时执行空语句操作；后者执行 $S2$ 语句。

5.1.3 循环结构

循环结构是指在一定条件下反复执行某部分代码的操作，是程序设计中最能发挥计算机特长的程序结构。循环结构分为当型循环结构和直到型循环结构，两者区别是，前者先判断条件是否成立，若成立则执行循环体；后者是先执行循环体再判断条件。循环结构的流程如图 5-3 所示。

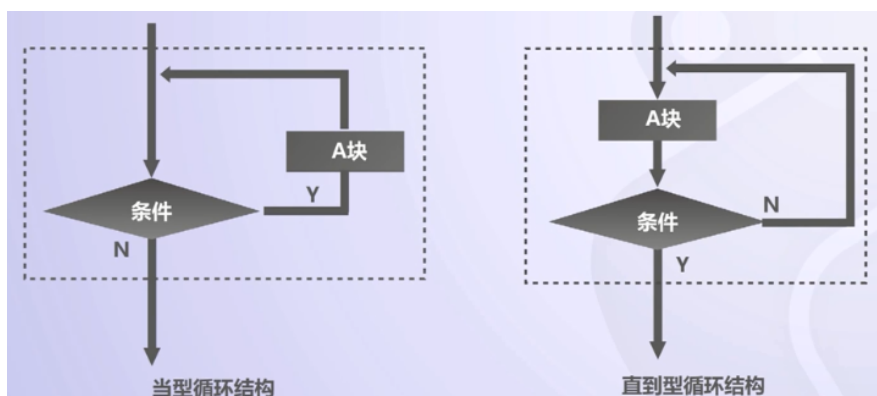


图 5-3 单选择结构与双选择结构

程序设计基本结构流程，如图 5-4 所示。

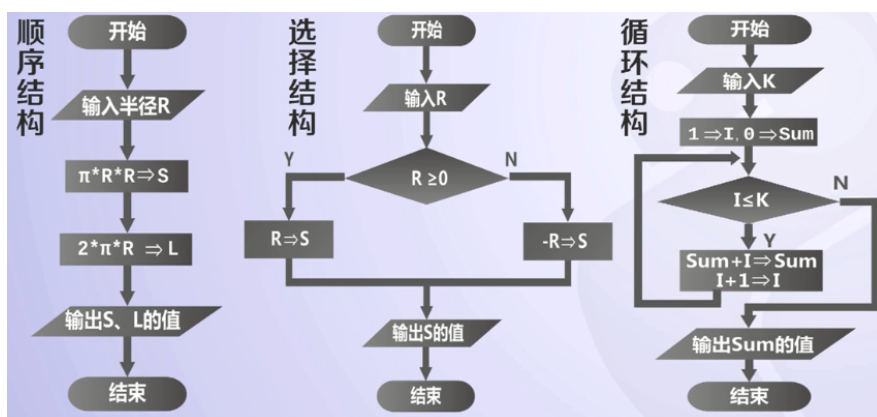


图 5-4 程序设计的三种基本结构流程

5.2 简单分支

例 5.1 PM2.5 指数分级例子：0-35 为优；35-75 为良；75-115 为轻度污染；115-150 为中度污染；150-250 为重度污染；250-500 为严重污染。现要设计程序，接受外部输入 PM2.5 值，输出空气质量提醒。

PM2.5 指数分级的流程如图 5-5 所示。

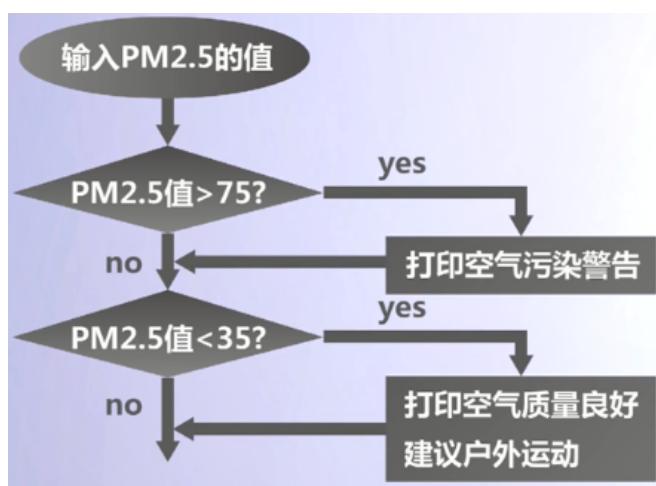


图 5-5 PM2.5 指数分级的流程

上述流程的 python 实现如下：

```
1 #pm25.py
2 def main():
3     PM=eval(input("what is today's PM2.5?"))
4     if PM>75:                                # 空气质量提醒
5         print("Unhealthy. Be careful!")
6     if PM<35:
7         print("Good. Go running!")
8
9 main()
```

if 语句格式如下

```
if <condition>:
    <body>
```

其中 <condition> 是条件表达式，<body> 是一个或多个语句序列。先判断 <condition> 条件：

- true，则执行 <body>，再转向下一条语句；
- false，则直接跳过 <body>，转向下一条语句；

简单条件基本形式：<expr> <relop> <expr>

<relop> 是关系操作符 <, <=, ==, >=, >, !=。使用 “=” 表示赋值语句；使用 “==” 表示等于，除数字外，字符或字符串也可以按照字典顺序用于条件比较。

<condition> 是布尔表达式，为 bool 类型，布尔值的真和假以字符 True 和 False 表示。

示例：

```
>>> 3<4
True
>>> 3*7<9
False
>>> "hello"=="Hello"
False
>>> "hello"<"Hello"
False
```

二分支语法结构如下：

```
if <condition>:
    <statements>
else:
    <statements>
```

Python 解释器首先评估 <condition>，如果 <condition> 是真的，if 下面的语句被执行；如果 <condition> 是假的，则 else 下面的语句被执行。

5.3 多分支

多分支决策是解决复杂问题的重要手段之一，一个三分支决策可以由两个二分支结构嵌套实现。

使用 if-else 描述多分支决策时，实现更多分支需要更多嵌套，影响程序的易读性，Python 使用 if-elif-else 描述多分支决策，简化分支结构的嵌套问题。

使用 if-elif-else 描述多分支决策的语法如下：

```
if <condition1>:
    <case1 statements>
elif <condition2>:
    <case2 statements>
elif <condition3>:
    <case3 statements>
.....
```

```
else:
    <default statements>
```

Python 轮流评估每个条件，来寻找条件为 True 的分支，并执行该分支下的语句；如果没有任何条件成立，else 下面的语句被进行，else 子句是可选的。

使用 if-elif-else 结构改进之前的 PM2.5 指数分级程序，如下：

```
1  #pm25_new.py
2  def main():
3      PM=eval(input("what is today's PM2.5?"))
4      if PM>250:                      # 空气质量提醒
5          print("Hazardous. Remain INDOORS!")
6      elif PM>150:
7          print("Very Unhealthy. Avoid prolonged exertion!")
8      elif PM>115:
9          print("Unhealthy. Limit prolonged exertion!")
10     elif PM>75:
11         print("Unhealthy. for sensitive group!")
12     elif PM>35:
13         print("Moderate. Go Walking!")
14     else:
15         print("Good. Go running!")
16
17  main()
```

5.4 异常处理

如果处理错误或特殊情况的分支语句过多，那么处理正常情况的主程序就会变得不清晰易读。

引入异常处理机制来解决程序运行时的错误，而不是显式检查算法的每一步是否成功。

Python 使用 try...except...来进行异常处理，基本格式如下：

```
try:
    <body>
except <ErrorType1>:
    <handler1>
```

```
except <ErrorType2>:  
    <handler2>  
except:  
    <handler0>
```

当 Python 解释器遇到一个 try 语句, 它会尝试执行 try 语句体 <body> 内的语句, 如果没有错误, 控制转到 try-except 后面的语句; 如果发生错误, Python 解释器会寻找一个符合该错误的异常语句, 然后执行处理代码。

```
1  #TryException.py  
2  def main():  
3      try:  
4          number1,number2=eval(input("Enter two numbers, separated by  
           ↪ a comma:"))  
5          result=number1/number2  
6      except ZeroDivisionError:  
7          print("Division by zero")  
8      except SyntaxError:  
9          print("A comma may be missing in the input")  
10     else:  
11         print("No exceptions, the result is ",result)  
12     finally:  
13         print("executing the final clause")  
14  
15  main()
```

运行实例:

```
>>> main()  
Enter two numbers, separated by a comma:3,4  
No exceptions, the result is  0.75  
executing the final clause  
>>> main()  
Enter two numbers, separated by a comma:2,0  
Division by zero  
executing the final clause
```

Try...except 可以捕捉任何类型的错误。对于二次方程, 还会有其他可能的错误, 如: 输入非数值类型 (NameError), 输入无效的表达式 (SyntaxError) 等。

此时可以用一个 try 语句配多个 except 来实现。

5.5 三者最大实例分析

例 5.2 分支结构设计举例。输入三个数，找出其中最大的数。

三者最大的 IPO 如下：

- 输入：三个数值
- 处理：三者最大算法
- 输出：打印最大值

程序设计策略：

S1：通盘比较，即将每一个值与其他所有值比较以确定最大值。

代码如下：

```
1 # comparison three numbers
2 x1,x2,x3=eval(input("Enter three numbers, separated by a
   ↪ comma:"))
3 if x1>x2 and x1>x3:
4     max=x1
5 elif x2>x1 and x2>x3:
6     max=x2
7 else:
8     max=x3
9 print("The maximum number is ",max)
```

存在的问题：

- 1) 目前只有三个值，比较简单，如果是五个值比较，表达式包含四个 and，比较复杂
- 2) 每个表达式结果没有被互相利用，效率低 (x1 与 x2 比较了两次)

S2：决策树，可以避免冗余比较，即先判断 $x1 > x2$ ，如果成立再判断 $x1 > x3$ ，否则判断 $x2 > x3$ 。

虽然效率高，但设计三个以上的方案，复杂性会爆炸性地增长。如设计一个决策树来判断四个值中的最大者，需要在 if-else 语句中嵌套 3 层结构和 8 个赋值语句。

代码如下：

```

1  # comparison three numbers
2  x1,x2,x3=eval(input("Enter three numbers, separated by a
    ↪ comma:"))
3  if x1>x2:
4      if x1>x3:
5          max=x1
6      else:
7          max=x3
8  else:
9      if x2>x3:
10         max=x2
11     else:
12         max=x3
13  print("The maximum number is ",max)

```

S3：顺序处理，借助循环结构，可以实现 n 个数最大者求值。

代码如下：

```

1  # find the maximum number
2  def maximum():
3      n=eval(input("How many numbers are there?: "))
4      max=eval(input("Enter the first number: ")) # 输入第一个
    ↪ 数字
5      for i in range(n-1): # 当前最大数连续与后面输入数字进行比
    ↪ 较
6          x=eval(input("Enter the next number: "))
7          if x>max:
8              max=x
9      print("The maximum number is: ",max)
10
11  maximum()

```

♣ **range** 函数用于产生一系列整数，语法如下：

```

range (start, end, step=1) # 不包含 end 的值
range (start, end) # 缺省 step 值为 1
range (end) # 缺省了 start 值为 0, step 为 1

```

S4 : Python 内置函数，使用 Python 内置的 max() 函数
代码如下：

```
1 def maximum():
2     x1,x2,x3=eval(input("please enter three numbers,
    ↪ separated by a comma: "))
3     print("The maximum number is: ",max(x1, x2, x3))
4
5 maximum()
```

程序设计思想：首先找到一个正确的计算问题，然后使其清晰、简洁、高效、优雅、可扩展。良好的算法和程序应该逻辑清晰，易于阅读和维护。

5.6 基本循环结构

5.6.1 for 循环

使用 for 语句可循环遍历整个序列的值，其语法如下：

```
for <var> in <sequence>:
    <body>
```

在 for 循环中，循环变量 var 遍历了 <sequence> 中的每一个值，循环的语句体为每个值执行一次。

```
>>> s = 'python'
>>> for c in s:
print(c)
p
y
t
h
o
n

>>> for i in range(3,11,2):
        print(i, end = ' ')
3 5 7 9
```

例 5.3 程序随机产生一个 0 300 间的整数，玩家竞猜，允许猜多次，系统给出“猜中”、“太大了”或“太小了”的提示。

```
1 # Filename: guessnum2.py
2 from random import randint
3 x = randint(0, 300)
4 for count in range(5):
5     print('Please input a number between 0~300: ')
6     digit = int(input())
7     if digit == x :
8         print('Bingo!')
9     elif digit > x:
10        print('Too large, please try again.')
11    else:
12        print('Too small, please try again.')
```

for 循环的缺点：明确循环的次数

Python 提供了另一种循环模式即无限循环，不需要提前知道循环次数，即我们提到的当型循环也叫条件循环。

5.6.2 while 循环

while 语句的语法如下：

```
while <condition>:
    <body>
```

while 语句中 <condition> 是布尔表达式，<body> 循环体是一条或多条语句。当条件 <condition> 为真时，循环体重复执行；当条件 <condition> 为假时，循环终止。

在 while 循环中，条件总是在循环顶部被判断，即在循环体执行之前，这种结构又被称为前测循环。

例 5.4 使用 while 循环完成从 0 到 n 的整数求和。

代码如下：

```
1 def sum_number(n):
2     sum,i=0,1
3     while i<n+1:
4         sum+=i
5         i+=1
6     print("the sum is: ",sum)
7
8 sum_number(10)
```

5.6.3 循环中的 break,continue 和 else

break 语句

break 语句终止当前循环，转而执行循环之后的语句。

```
# breakpro.py
sumA ,i=0,1
while True:
    sumA += i
    i += 1
    if sumA > 10:
        break
print('i={},sum={}'.format(i, sumA))
```

运行结果如下：

```
i=6,sum=15
```

continue 语句

在 while 和 for 循环中，continue 语句的作用：

- 停止当前循环，重新进入循环
- while 循环则判断循环条件是否满足
- for 循环则判断迭代是否已经结束

```
# 循环中的 continue
sumA ,i=0,1
while i <= 5:
    sumA += i
    i += 1
    if i == 3:
        continue
    print('i={},sum={}'.format(i,sumA))
```

运行结果如下：

```
i=2,sum=1
i=4,sum=6
i=5,sum=10
i=6,sum=15
```

```
# 循环中的 break
while i <= 5:
    sumA += i
    if i == 3:
        break
    print('i={},sum={}'.format(i,sumA))
    i += 1
```

运行结果如下：

```
i=1,sum=1
i=2,sum=3
```

♠ continue 语句和 break 语句的区别是：continue 语句只结束本次循环，而不终止整个循环的执行；而 break 语句则是结束整个循环过程，不再判断执行循环的条件是否成立。

循环中的 else 语句

循环中的 else:

- 如果循环代码从 break 处终止，跳出循环

- 正常结束循环，则执行 else 中代码

```
# prime.py
from math import sqrt
num = int(input('Please enter a number: '))
j = 2
while j <= int(sqrt(num)):
    if num % j == 0:
        print('{:d} is not a prime.'.format(num))
        break
    j += 1
else:
    print('{:d} is a prime.'.format(num))
```

5.7 通用循环构造方法

5.7.1 交互式循环

交互式循环是无限循环的一种，允许用户通过交互的方式重复程序的特定部分。

让我们以交互循环的视角重新审视求平均数程序，伪码如下：

```
初始化 sum 为 0
初始化 count 为 0
初始化 moredata 为"yes"
当 moredata 值为"yes"时
    输入数字 x
    将 x 加入 sum
    count 值加 1
    询问用户是否还有 moredata 需要处理
输出 sum/count
```

交互式循环求平均数程序代码如下：

```
1  def main():
2      sum,count=0,0
3      moredata="yes"
4      while moredata=="yes":
5          x=eval(input("Enter a number>> "))
6          sum+=x
7          count+=1
8          moredata=input("Do you have more numbers (yes or no)?")
9      print("The average of the numbers is {}".format(sum/count))
10
11  main()
```

5.7.2 哨兵循环

哨兵循环是执行循环直到遇到特定的值，循环语句才终止执行的循环结构设计方法。

哨兵循环是求平均数的更好方案，思路如下：

- 设定一个哨兵值作为循环终止的标志
- 任何值都可以做哨兵，但要与实际数据有所区别

哨兵循环求平均数程序代码如下：

```
1  def main():
2      sum,count=0,0
3      x=eval(input("Enter a number(negative to quit)>> "))
4      while x>=0: #0 作为哨兵数据
5          sum+=x
6          count+=1
7          x=eval(input("Enter a number(negative to quit)>> "))
8      print("\n The average of the numbers is {}".format(sum/count))
9
10  main()
```

5.7.3 文件循环

面向文件的方法是数据处理的典型应用。之前求平均数的数字都是用户输入的，如果几百个数求平均，输入困难且容易出错；可以事先将数据录入到文

件中，然后将这个文件作为程序的输入，避免人工输入的麻烦，便于编辑修改。

```
1  def main():
2      filename=input("what's the file's path? ")
3      infile=open(filename,'r')
4      sum,count=0,0
5      for line in infile:
6          sum+=eval(line)
7          count+=1
8      print("\n The average of the numbers is {}".format(sum/count))
9
10 main()
```

5.8 死循环、后测循环和半路循环

5.8.1 死循环

死循环并非一无是处,c 语言中死循环 `while true` 或 `while 1` 是单片机编程的普遍用法，死循环一直运行等待中断程序发生，然后去处理中断程序。

在 python 中我们也可以利用死循环完成特定功能。

```
1  while True:
2      try:
3          num1 = int(input('Enter the first number: '))
4          num2 = int(input('Enter the second number: '))
5          print(num1 / num2)
6          break
7      except ValueError:
8          print('Please input a digit!')
9      except ZeroDivisionError:
10         print('The second number cannot be zero!')
```

5.8.2 后测循环

条件判断在循环体后面，称之为后测循环。后测循环至少执行一次循环体。

Python 没有后测循环语句，但可以通过 `while` 间接实现，思想是设计一个循环条件，直接进入循环体，循环至少执行一次，相当于后测循环。

```
number=-1
while number<0:
    number=eval(input("Enter a positive number: "))
```

break 语句也可以用来实现后测循环。

```
#while 语句体永远执行, if 条件决定循环退出
while True:
    number=eval(input("Enter a positive number: "))
    if number>0:break #if 语句体只包含一个语句时, break 可以跟 if
    ↪ 在同一行
```

5.8.3 半途循环

运用 break 中途退出循环, 循环出口在循环体中部, 被称为半途循环。

```
while True:
    number=eval(input("Enter a positive number: "))
    if number>0:break #break 跳出循环
```

应避免在一个循环体内使用过多的 break 语句, 因为当循环有多个出口的时候, 程序逻辑就显得不够清晰了。

5.9 布尔表达式

条件语句和循环语句都使用布尔表达式作为条件。布尔值为真或假, 以 False 和 True 表示, 前面经常使用布尔表达式比较两个值, 如: while x>=0。

简单条件在复杂决策情况下存在一定缺陷, 例如, 确定两个点是否是在同一个位置, 即是否有相同的 x 坐标和 y 坐标, 下面是处理的代码片段:

```
if p1.getX()==p2.getX():
    if p1.getY()==p2.getY():
        # 两点相同
    else:
        # 两点不同
else:
    # 两点不同
```

上述判断过于复杂, Python 提供了更简单的布尔操作符来构建表达式。

5.9.1 布尔操作符

布尔操作符

布尔操作符：and, or 和 not

布尔运算符 and 和 or 用于组合两个布尔表达式，并产生一个布尔结果：

```
<expr> and <expr>
```

```
<expr> or <expr>
```

not 运算符是一个一元运算符，用来对一个布尔表达式取反

```
not <expr>
```

Python 中布尔操作符的优先级，从高分到低分依次是 not、and 最低是 or。例如 $a \text{ or } \text{not } b \text{ and } c \iff (a \text{ or } ((\text{not } b) \text{ and } c))$

使用 and 操作符改进之前比较两个点相同的例子

```
if p1.getX()==p2.getX() and p1.getY()==p2.getY():
    # 两点相同
else:
    # 两点不同
```

5.9.2 布尔代数

布尔表达式遵循特定的代数定律，这些规律被称为布尔逻辑或布尔代数。布尔代数规则如下表

表 5-1 布尔代数规则

Algebra	Boolean Algebra
$a * 0 = 0$	$a \text{ and } \text{false} == \text{false}$
$a * 1 = a$	$a \text{ and } \text{true} == a$
$a + 0 = a$	$a \text{ or } \text{false} == a$

当 0 和 1 对应 false 和 true 时，and 与乘法相似；or 与加法相似。

任何值和 true 进行“or”操作都是真：

```
a or true == true
```

and 和 or 操作符都符合分配率:

```
a or (b and c)==(a or b) and (a or c)
a and (b or c)==(a and b) or (a and c)
```

not 操作符具有负负抵消的特性:

```
not (not a)==a
```

布尔代数符合德摩根定律, not 放进表达式后, and 和 or 运算符之间发生的变化:

```
not (a or b)==(not a ) and (not b)
not (a and b)==(not a ) or (not b)
```

Python 的条件运算符 (即 ==) 总是在与一个 bool 类型的值进行比较!

布尔 True 和 False 来代表布尔值的真和假。

对于数字 (整型和浮点型) 的零值被认为是 false, 任何非零值都是 true。

bool 类型仅仅是一个特殊的整数, 可以通过计算表达式 True + True 的值来测试一下。

对于序列类型来说, 一个空序列被解释为假, 而任何非空序列被指示为真。

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(3)
True
>>> bool("Y")
True
>>> bool("Hello Python")
True
>>> bool("")
False
>>> bool([])
False
>>> bool([1,2,3])
True
```


Python 的布尔运算符是短路运算符，即

(表达式 1) and (表达式 2) # 如果表达式 1 为假，则表达式 2 不会进行运算，即表达式 2 “被短路”；
(表达式 1) or (表达式 2) # 如果表达式 1 为真，则表达式 2 不会进行运算，即表达式 2 “被短路”

如果用户仅仅简单敲下回车键，可以使用方括号中的值作为默认值：

```
ans=input("what flavor do you want [vanilla]: ")
if ans != "":
    flavor=ans
else:
    flavor="vanilla"
```

可以简化如下：

```
ans=input("what flavor do you want [vanilla]: ")
if ans:
    flavor=ans
else:
    flavor="vanilla"
```

更简洁的表达形式

```
ans=input("what flavor do you want [vanilla]: ")
flavor=ans or "vanilla"
```

进一步简化：

```
flavor=input("what flavor do you want [vanilla]: ") or
    "vanilla"
```

6 Python 编程之代码复用

6.1 函数

函数：完成特定功能的一个语句组，通过调用函数名来完成语句组的功能。
为函数提供不同的参数，可以实现对不同数据的处理。
函数可以反馈结果。

6.1.1 函数的定义

函数分类：

- 自定义函数：用户自己编写的
- 系统自带函数：Python 内嵌的函数（如 `abs()`、`eval()`）、Python 标准库中的函数（如 `math` 库中的 `sqrt()`）、图形库中的方法（如 `myPoint.getX()`）等

使用函数目的：降低编程的难度；代码重用。

函数定义

```
def FunctionName([arguments]):  
    <body>  
    FunctionSuite
```

- 函数名 `FunctionName`：可以是任何有效的 Python 标识符
- 参数列表 `[arguments]`：是调用函数时传递给它的值（可以由多个，一个或者零个参数组成，当有多个参数时，各个参数用逗号分隔）
 - 1) 形式参数：定义函数时，函数名后面圆括号中的变量，简称“形参”。形参只在函数内部有效。
 - 2) 实际参数：调用函数时，函数名后面圆括号中的变量，简称“实参”。
- 函数体 `<body>`：函数被调用时执行的代码，由一个或多个语句组成。

例 6.1 定义一个函数：求输入值的两倍并返回。

```
def addMeToMe(x):  
    x+=x  
    return x
```

`return` 语句：结束函数调用，并将结果返回给调用者

`return` 语句是可选的，可出现在函数体的任意位置。没有 `return` 语句，函数在函数体结束位置将控制权返回给调用方。

例 6.2 综合例子：编写 `happy()`, `sing()` 函数，为 Mike、Lily 二人唱生日歌。

```
1  # 生日程序
2  def happy():
3      print("Happy birthday to you!")
4
5  def sing(person):
6      happy()
7      happy()
8      print("Happy birthday, dear", person+"!")
9      happy()
10
11 def singbirthday():
12     sing("Mike")
13     print() # 打印空行
14     sing("Lily")
15
16 singbirthday()
```

打印歌词如下：

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Mike!
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lily!
Happy birthday to you!
```

6.1.2 函数的调用和返回

函数的调用

函数调用之前必须先定义。函数名加上函数运算符，一对小括号

- 1 括号之间是所有可选的参数
- 2 即使没有参数，小括号也不能省略

函数调用执行的四个步骤：

- 1 调用程序在调用处暂停执行
- 2 函数的形参在调用时被赋值为实参
- 3 执行函数体
- 4 函数被调用结束，给出返回值

例 6.1 中的函数调用如下：

```
addMetoMe(3)
```

分析例 6.2 中的节生日歌词程序 `singbirthday()` 中部分程序，`happy()` 和 `sing()` 的完整调用过程分别如图 6-1、6-2 所示。

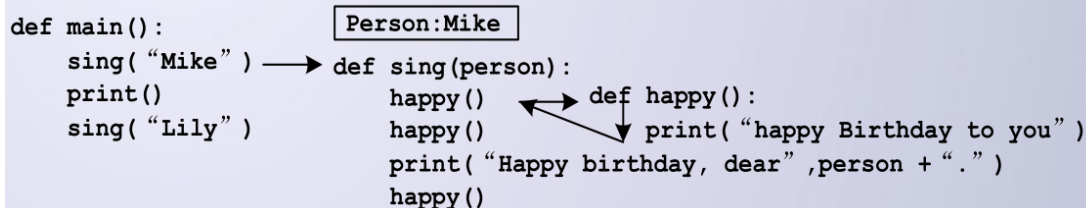


图 6-1 `happy()` 完整调用过程图

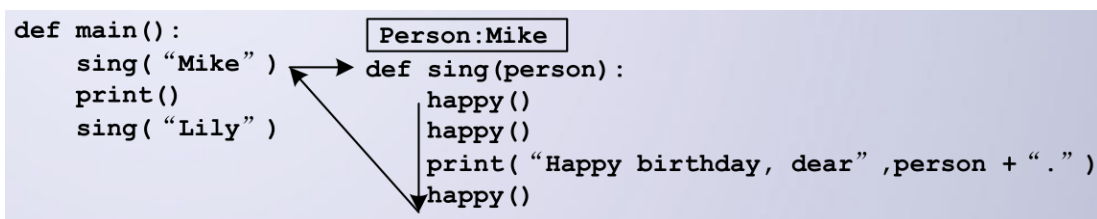


图 6-2 `sing()` 完整调用过程图

函数的返回值

`return` 语句：程序退出该函数，并返回到函数被调用的地方，返回值传递给调用程序。

Python 函数的返回值有两种形式：

- 1 返回一个值
- 2 返回多个值

无返回值的 `return` 语句等价于 `return None`。`None` 是表示没有任何东西的特殊类型。

返回值可以是一个变量，也可以是一个表达式。

例 6.1 中的函数返回值可写成：

```
def addMeToMe(x):  
    return (x*2)
```

例 6.3 编写程序计算三角形周长。

```
1  import math  
2  def square(x): # 定义平方函数  
3      return (x*x)  
4  
5  def distance(x1,y1,x2,y2): # 定义求两点距离函数  
6      dist=math.sqrt(square(x1-x2)+square(y1-y2))  
7      return dist  
8  
9  def isTriangle(x1,y1,x2,y2,x3,y3): # 判断三点是否构成三角形  
10     flag=((x1-x2)*(y3-y2)-(x3-x2)*(y1-y2))!=0  
11     return flag  
12  
13 def main():  
14     print("Please enter (x,y) of three points in turn: ")  
15     x1,y1=eval(input("Point1: (x,y)= ")) # 输入三点坐标  
16     x2,y2=eval(input("Point2: (x,y)= "))  
17     x3,y3=eval(input("Point3: (x,y)= "))  
18     if (isTriangle(x1,y1,x2,y2,x3,y3)): # 计算三角形周长  
19         perim=distance(x1,y1,x2,y2)+distance(x1,y1,x3,y3)  
20             +distance(x2,y2,x3,y3)  
21         print("The perimeter of the triangle is:  
22             ↪ {0:0.2f}".format(perim))  
23     else:  
24         print("Kidding me? This is not a triangle!")  
25     main()
```

运行结果：

```
>>> main()
Please enter (x,y) of three points in turn:
Point1: (x,y)= 1,1
Point2: (x,y)= 2,2
Point3: (x,y)= 3,3
Kidding me? This is not a triangle!

>>> main()
Please enter (x,y) of three points in turn:
Point1: (x,y)= 1,1
Point2: (x,y)= 2,4
Point3: (x,y)= 4,2
The perimeter of the triangle is: 9.15
```

使用 return 语句返回多个值。

例 6.4 计算两个数的加法和减法

```
1  def SumDiff(x,y):
2      Sum=x+y
3      diff=x-y
4      return Sum,diff
5
6  number1,number2=eval(input("Please enter two numbers
   ↪ (number1,number2): "))
7  s,d= SumDiff(number1,number2)
8  print("The sum is ",s,"and the difference is",d)
```

对于多返回值的函数，根据变量的位置来赋值。s 将获得 return 的第一个返回值 sum；d 将获得第二个返回值 diff。

6.1.3 改变参数值的函数

函数通过参数与调用程序传递信息。

例 6.5 银行账户计算利率——账户余额计算利息的函数。

```
1 def addInterest(balance,rate):
2     return balance*(1+rate)
3 def main():
4     amount = 1000
5     rate = 0.05
6     amount=addInterest(amount,rate)
7     print(amount)
8
9 main()
```

总结：Python 的参数是通过值来传递的。如果变量是可变对象（如列表或者图形对象），返回到调用程序后，该对象会呈现被修改后的状态。

6.2 函数与递归

6.2.1 函数和程序结构

函数可以简化程序，函数可以使程序模块化；用函数将较长的程序分割成短小的程序段，可以方便理解。

例 6.6 有如下程序：

```
1 print("This program plots the growth of a 10-year investment.")
2 # 输入本金和利率
3 principal=eval(input("Enter the initial principal: "))
4 apr=eval(input("Enter the annualized interst rate:" ))
5 # 建立一个图表，绘制每一年银行账户的增长数据
6 for year in range(1,11):
7     principal=principal*(1+apr)
8     print("%2d"%year,end=' ')
9     # 计算星号数量
10    total=int(principal*4/1000)
11    print("*"*total)
12 print("0.0K    2.5K    5.0K    7.5K    10.0K")
```

将上述代码中部分功能移出作为独立函数：

```

# 计算星号数量
def calculateNum(principal):
    total=int(principal*4/1000)
    return total

# 为每一年绘制星号增长图
def createTable(principal,apr):
    for year in range(1,11):
        principal=principal*(1+apr)
        print("%2d"%year,end=' ')
        total=calculateNum(principal)
        print(" "*total)
    print("0.0K      2.5K      5.0K      7.5K      10.0K")

# 主程序
def main():
    print("This program plots the growth of a 10-year
    ↪ investment.")# 输入本金和利率
    principal=eval(input("Enter the initial principal: "))
    apr=eval(input("Enter the annualized interst rate:" ))
    createTable(principal,apr) # 建立图表

```

程序运行结果为:

```
This program plots the growth of a 10-year investment.
```

```
Enter the initial principal: 1000
```

```
Enter the annualized interst rate:0.2
```

```
1****
2*****
3*****
```



```

4*****
5*****
6*****
7*****
8*****
9*****
10*****
0.0K      2.5K      5.0K      7.5K      10.0K

```

程序结构化代码如下：

```

1  # 计算星号数量
2  def calculateNum(principal):
3      total=int(principal*4/1000)
4      return total
5
6  # 为每一年绘制星号增长图
7  def createTable(principal,apr):
8      for year in range(1,11):
9          principal=principal*(1+apr)
10         print("%2d"%year,end=' ')
11         total=calculateNum(principal)
12         print(" "*total)
13     print("0.0K      2.5K      5.0K      7.5K      10.0K")
14
15 # 主程序
16 def main():
17     print("This program plots the growth of a 10-year
18         ↪ investment.")# 输入本金和利率
19     principal=eval(input("Enter the initial principal: "))
20     apr=eval(input("Enter the annualized interst rate:" ))
21     createTable(principal,apr) # 建立图表
22
23 main()

```

使用函数的思想编写程序，可以大大增加程序的模块化程度，增强整个程序的可读性。

6.2.2 递归

递归：函数定义中使用函数自身的方法。递归是最能表现计算思维的算法之一。

例 6.7 利用递归编程生成斐波那契数列。斐波那契数列的递归公式为：

$$\begin{cases} F(0) = 0, F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad n \geq 2, n \in N^* \end{cases}$$

斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……

递归必须要有边界条件，即停止递归的条件。本例中递归停止的条件为：

```
n == 0 or n == 1
```

常规方法：

```
1 # the Fibonacci number
2 def fib(n):
3     a, b = 0, 1
4     count = 1
5     while count < n:
6         a, b = b, a+b
7         count = count + 1
8         print(a)
9
10 n=eval(input("Please enter a integer(>1): "))
11 fib(n)
```

运用递归思想：

```
1 # the nth Fibonacci number
2 def fib(n):
3     if n == 0 or n == 1:
4         return n
5     else:
6         return fib(n-1)+fib(n-2)
```

递归的代码更简洁，更符合自然逻辑，更容易理解。

递归的执行：逐层递归调用；遇到边界条件停止递归；逐层返回调用至最初层；系统资源消耗比循环大。

例 6.8 利用递归编程计算 n 阶乘：

$$n! = \begin{cases} 1, n = 0 \\ n(n-1)!, n \geq 1 \end{cases}$$

代码如下：

```
def fact(n):
    if n==0: return 1
    else: return n*fact(n-1)
```

例 6.9 汉诺塔游戏。三个塔座 A、B、C 上各有一根针，如图 6-3 所示，通过 B 把 N 个盘子从 A 针移动到 C 针上，移动时必须遵循下列规则：

- (1) 圆盘可以插入在 A、B 或 C 塔座的针上
- (2) 每次只能移动一个圆盘
- (3) 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上

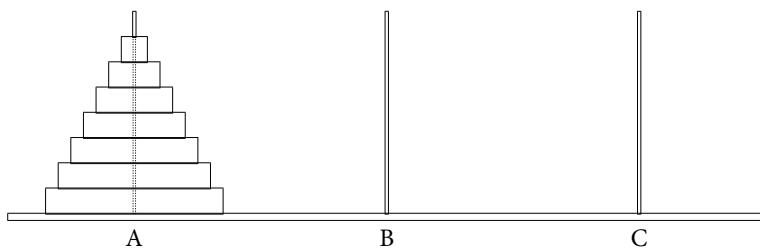


图 6-3 汉诺塔游戏

分析：将 n 个盘子从 A 针移到 C 针可以分解为三个步骤：

- (1) 将 A 上 $n-1$ 个盘子移到 B 针上（借助 C 针）；
- (2) 把 A 针上剩下的一个盘子移到 C 针上；
- (3) 将 $n-1$ 个盘子从 B 针移到 C 针上（借助 A 针）。

代码如下：

```
1 # Hanoi.py
2 def hanoi(a,b,c,n):
3     if n==1:
4         print(a,'->',c)
5     else:
6         hanoi(a,c,b,n-1)
7         print(a,'->', c)
8         hanoi(b,a,c,n-1)
9 hanoi('a','b','c',4)
```

6.3 函数实例分析

例 6.10 利用 turtle 库绘制并填充一个五角星。

代码如下：

```
1  from turtle import *
2  color('red', 'yellow')
3  begin_fill()
4  while True:
5      forward(200)
6      left(170)
7      if abs(pos()) < 1:
8          break
9  end_fill()
10 done()
```

运行结果如下：

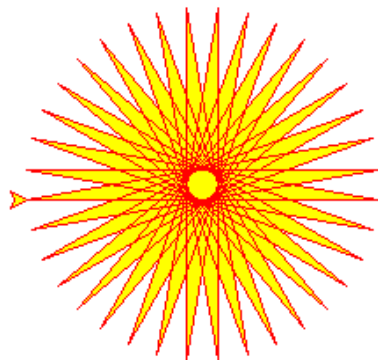


图 6-4 star

例 6.11 树的绘制，如下图 6-5 所示。

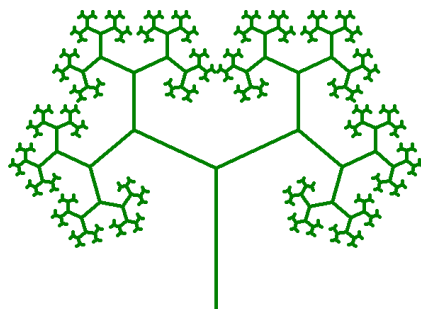


图 6-5 tree

分析：观察树的图案，这是一个对称树，从主杆出发以一定角度向左向右生成对称的枝丫，而每一棵枝杈上以相同的角度生成更小的左右枝杈，如此往复。联系我们所学过的内容，很容易想到可以利用递归程序实现，程序代码如下：

```
from turtle import Turtle, mainloop

def tree(plist, l, a, f):
    """ plist is list of pens
    l is length of branch
    a is half of the angle between 2 branches
    f is factor by which branch is shortened
    from level to level."""
    if l > 5: #
        lst = []
        for p in plist:
            p.forward(l) # 沿着当前的方向画画 Move the turtle
                ↳ forward by the specified distance, in the
                ↳ direction the turtle is headed.
            q = p.clone() # Create and return a clone of the
                ↳ turtle with same position, heading and turtle
                ↳ properties.
            p.left(a) # Turn turtle left by angle units
            q.right(a) # turn turtle right by angle units, nits
                ↳ are by default degrees, but can be set via the
                ↳ degrees() and radians() functions.
            lst.append(p) # 将元素增加到列表的最后
            lst.append(q)
        tree(lst, l*f, a, f)

def main():
    p = Turtle()
    p.color("green")
    p.pensize(5)
    #p.setundobuffer(None)
    p.hideturtle() # Make the turtle invisible. It's a good
        ↳ idea to do this while you're in the middle of doing
        ↳ some complex drawing,
```

```
#because hiding the turtle speeds up the drawing
↳ observably.
#p.speed(10)
#p.getscreen().tracer(1,0) #Return the TurtleScreen object
↳ the turtle is drawing on.
p.speed(10)
#TurtleScreen methods can then be called for that object.
p.left(90)# Turn turtle left by angle units. direction 调整
↳ 画笔

p.penup() #Pull the pen up - no drawing when moving.
p.goto(0,-200)#Move turtle to an absolute position. If the
↳ pen is down, draw line. Do not change the turtle's
↳ orientation.
p.pendown()# Pull the pen down - drawing when moving. 这三
↳ 条语句是一个组合相当于先把笔收起来再移动到指定位置，再把笔
↳ 放下开始画
# 否则 turtle 一移动就会自动的把线画出来

#t = tree([p], 200, 65, 0.6375)
t = tree([p], 200, 65, 0.6375)

main()
```

如何画出多棵树，甚至整片森林呢？答案很简单，只要在画每棵树之前调整画笔的位置，调用画树程序，就可以从新位置生成一颗新树了。

利用模块化的函数思想，调整代码：将每棵树的绘制以 `maketree` 函数封装，参数 `x,y` 为画树的起点位置即树根位置，在 `main` 函数中只要以不同的参数设置来调用 `maketree` 函数就可以完成多棵树的绘制了。

7 Python 编程之组合类型

7.1 文件

7.1.1 文件的基础

文件是存储在外部介质上的数据或信息的集合。如程序中的源程序、数据中保存着数据以及图像中的像素数据等。文件是有序的数据序列。

编码是信息从一种形式转换为另一种形式的过程。常用的编码：

- **Unicode**：跨语言、跨平台进行文本转换和处理；对每种语言中字符设定统一且唯一的二进制编码；每个字符两个字节长；65536 个字符的编码空间。如“严”的 Unicode 的十六进制数为 4E25
- **UTF-8 编码**：可变长度的 Unicode 的实现方式。“严”的十六进制数为 E4B8A5
- **GBK**(《汉字内码扩展规范》) 编码，双字节编码

文件数据：

- 文本文件：以 ASCII 码方式存储的文件
- 二进制文件，优点：节省空间、采用二进制无格式存储、表示更为精确

注意：

- 文本文件是基于字符定长的 ASCII
- 二进制文件编码是变长的，灵活利用率要高
- 不同的二进制文件解码方式是不同的

7.1.2 文件的基本处理

文件的基本处理包括：

- 打开文件：建立磁盘上的文件与程序中的对象相关联；过相关的文件对象获得
- 文件操作：读取、写入、定位，其他：追加、计算等
- 关闭文件：切断文件与程序的联系；写入磁盘，并释放文件缓冲区

打开文件

```
file_obj = open(filename, mode)
```

其中：filename 为文件名，mode 为打开模式（可选参数），默认值为 r。
open() 函数的几种打开模式如下表 7-1。

表 7-1 open() 函数的几种打开模式

Mode	Function
r	以读模式打开
w	以写模式打开（清空原内容）
a	以追加模式打开（从 EOF 开始，必要时创建新文件）
r+	以读写模式打开
w+	以读写模式打开（清空原内容）
a+	以读和追加模式打开
rb	以二进制读模式打开
wb	以二进制写模式打开（参见 w）
ab	以二进制追加模式打开（参见 a）
rb+	以二进制读写模式打开（参见 r+）
wb+	以二进制读写模式打开（参见 w+）
ab+	以二进制读写模式打开（参见 a+）

open() 函数返回一个文件（file）对象，文件对象可迭代。

例 7.1 打开一个名为“numbers.dat”的文本文件

```
>>> infile = open ("numbers.dat", "r")
```

例 7.2 打开一个名为“music.mp3”的音频文件

```
>>>infile = open ("music.mp3", "rb")
```

读取文件

read() 返回值为包含整个文件内容的一个字符串。

readline() 返回值为文件下一行内容的字符串。

readlines() 返回值为整个文件内容的列表，每项是以换行符为结尾的一行字符串。

例 7.3 将文件内容输出到屏幕上。


```
def main():
    filename=input("Enter filepath: ")
    infile=open(filename,"r")
    data=infile.read()
    print(data)

main()
```

例 7.4 输出文件前 5 行内容。

```
def main():
    filename=input("Enter filepath: ")
    infile=open(filename,"r")
    for i in range(5):
        line=infile.readline()
        print(line[:-1])

main()
```

写入文件

从计算机内存向文件写入数据。

`write()`: 把含有本文数据或二进制数据块的字符串写入文件中。

`writelines()`: 针对列表操作, 接受一个字符串列表作为参数, 将它们写入文件。

例 7.5 将一个字符串写入文件。

```
with open('write.txt', 'w') as f:
    f.write('Hello, World!')
    f.close()
```

例 7.6 将文件 `companies.txt` 的字符串前加上序号 1、2、3、...后写到另一个文件 `scompanies.txt` 中。

```
with open('companies.txt') as f1:
    cNames = f1.readlines()
    for i in range(0, len(cNames)):
        cNames[i] = str(i+1) + ' ' + cNames[i]
with open('scompanies.txt', 'w') as f2:
    f2.writelines(cNames)
```

例 7.7 将“Tencent Technology Company Limited”添加到 companies.txt 中。

```
s = 'Tencent Technology Company Limited'
with open('companies.txt', 'a+') as f1:
    f1.writelines('\n') # 换行
    f1.writelines(s)    # 写入数据
    cNames = f1.readlines() # 读取全部数据，由于文件指针处于
    ↪ 文件末尾，此时读取为空值
    print(cNames)
```

文件指针

```
file_obj.seek(offset, whence=0)
```

- 在文件中移动文件指针，从 whence（0 表示文件头部，1 表示当前位置，2 表示文件尾部）偏移 offset 个字节
- whence 参数可选，默认值为 0

```
1 s='good jod!'
2 with open('companies.txt', 'a+') as f2:
3     f2.writelines('\n') # 换行
4     f2.writelines(s)    # 写入数据
5     f2.seek(0)          # 写入数据后将指针移至文件头部
6     print(f2.readlines()) # 从文件头部开始读取数据
```

7.1.3 文件实例一

例 7.8 根据文件 data.txt 中的数据,使用 turtle 库来动态绘制图形路径。数据结构如下图 7-1 所示,其含义:第一列表示前进像素;第二列表示方向(0 表示向左转,1 表示向右转);第三列表示转动角度;后三列表示颜色 rgb。

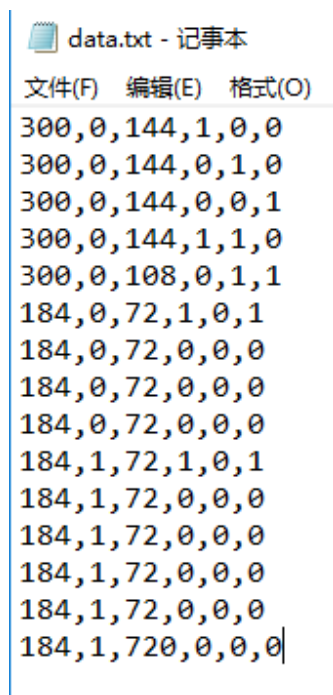


图 7-1 data 文件数据结构

该问题的 IPO 模式可以描述为:

- 输入: 数据文件
- 处理: 读取数据文件,并根据数据内容和要求绘制路径。
- 输出: 构建窗口,并输出图形

程序实现的具体过程为:

- (1) 使用 import 命令为程序引入 turtle 库
- (2) 设置窗口信息和 Turtle 画笔
- (3) 读取数据文件到列表中
- (4) 根据每一条数据记录进行绘制

```

1  from turtle import *
2
3  # 设置窗口信息
4  title(" 数据驱动的动态路径绘制")
5  setup(800,600,0,0)
6  # 设置画笔
7  pen=Turtle()
8  pen.color('red')
9  pen.width(5)
10 pen.shape("turtle")
11 pen.speed(2)
12
13 # 读取数据文件到列表 result 中
14 result=[]
15 file=open('data.txt')
16 for line in file:
17     result.append(list(map(float,line.split(','))))
18 print(result)
19
20 # 根据每一条数据记录进行绘制
21 for i in range(len(result)):
22     pen.color(result[i][3],result[i][4],result[i][5])
23     pen.fd(result[i][0])
24     if result[i][1]:
25         pen.rt(result[i][2])
26     else:
27         pen.lt(result[i][2])

```

运行结果如下：

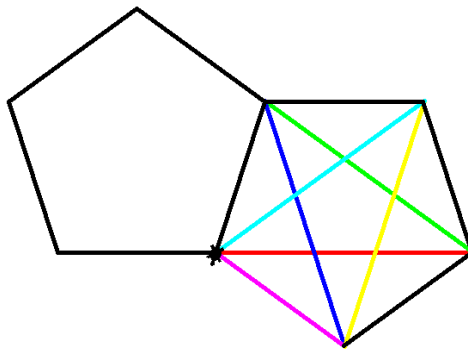


图 7-2 图形路径

7.1.4 文件实例二

例 7.9 多文件读写例子。编写程序将电话簿 TeleAddressBook.txt 和电子邮件 EmailAddressBook.txt 合并为一个完整的 AddressBook.txt。源数据文件格式如图 7-6 所示，目标文件如图 7-7 所示。

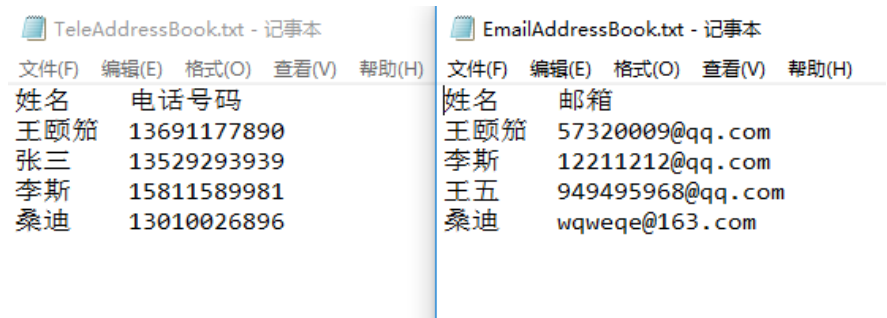


图 7-3 源文件数据结构

姓名	电话	邮箱
王筋颐	57320009@qq.com	13691177890
李斯	12211212@qq.com	15811589981
王五	949495968@qq.com	-----
桑迪	wqweqe@163.com	13010026896
张三	-----	13529293939

图 7-4 合并后文件数据结构

- 该问题的 IPO 模式可以描述为：
- 输入：电话簿、邮箱地址文件
 - 处理：将两个文件内容进行合并
 - 输出：合并后包含电话和邮箱地址簿的文件
- 程序实现的具体过程为：
- (1) 打开文件、读取文件
 - (2) 分别获取文件中的信息到两个列表 1 和列表 2 中
 - (3) 建立合并用的空列表 3，完成列表 1 和 2 的信息合并操作到列表 3 中，
- 具体实现方法：
- 生成信息表头
 - 遍历列表 1 中的人，创立列表 3 的信息
 - 处理列表 2 中剩余人的信息，合并到列表 3 中
- (4) 将列表 3 中的信息写入到新文件
 - (5) 关闭所有打开文件

```

femail=open("EmailAddressBook.txt",'rb')
ftele=open("TeleAddressBook.txt",'rb')

femail.readline() # 跳过“姓名”行
ftele.readline()
line_email=femail.readlines() # 读文件
line_tele=ftele.readlines()

list1_name,list1_tele,list2_name,list2_email,list_new=[],[],[],[],[]
↪ # 建立空列表用于存储姓名、电话、Email

# 获取 TeleAddressBook 中的姓名、电话信息
for line in line_tele:
    elements=line.split()
    list1_name.append(str(elements[0].decode('gbk')))
    list1_tele.append(str(elements[1].decode('gbk')))

# 获取 TeleAddressBook 中的姓名、邮件信息
for line in line_email:
    elements=line.split()
    list2_name.append(str(elements[0].decode('gbk')))
    list2_email.append(str(elements[1].decode('gbk')))

list_new.append(' 姓名\t 电话 \t 邮箱\n')

for i in range(len(list1_name)):
    s=''
    if list1_name[i] in list2_name:
        # 找到姓名列表 1 对应列表 2 中的姓名索引位置
        j=list2_name.index(list1_name[i])
        s='\t'.join([list1_name[i],list1_tele[i],list2_email[j]])
        s+='\n'
    else:
        s='\t'.join([list1_name[i],list1_tele[i],str(' -----
        ↪ ')])
        s+='\n'

```

```

        list_new.append(s)

# 处理姓名列表 2 中剩余的姓名
for i in range(len(list1_name)):
    s=''
    if list1_name[i] not in list2_name:
        s='\t'.join([list2_name[i],str(' -----
        ↪ '),list2_email[j]])
        s+='\n'
        list_new.append(s)

# 将合并后的输几局写入文件
with open("AddressBook.txt",'w') as f:
    f.writelines(list_new)
ftele.close()
femail.close()
f.close()
print("The addressBooks are merged!")

```

7.2 字典

为什么要使用字典？我们先来看看这样一个例子：

例 7.10 某公司人事部门让技术部门用 Python 构建一个简易的员工信息表，包含员工的姓名和工资信息。根据信息表查询员工牛云的工资。

代码如下：

```

# info.py
names = ['Wangdachui', 'Niuyun', 'Linling', 'Tianqi']
salaries = [3000, 2000, 4500, 8000]
print(salaries[names.index('Niuyun')])

```

从上面代码可以看出，查询过程似乎比较“繁琐”，是否可以考虑使用如下代码更加方便？

```

salaries['Niuyun']

```

7.2.1 字典的基础

字典是针对非序列集合而提供的一种数据类型。字典是一种映射，即通过任意键值查找集合中值信息的过程。

字典是键值对的集合，该集合以键为索引，同一个键信息对应一个值。

字典类型与序列类型的区别：

- 存取和访问方式不同
- 键的类型不同
 - 序列类型只能用数字类型的键
 - 字典类型可以用其他对象类型作键
- 排列方式不同
 - 序列类型保持了元素的相对关系
 - 而字典中的数据是无序排列的。
- 映射方式不同
 - 序列类型通过地址映射到值
 - 字典类型通过键直接映射到值

创建字典：

- 直接
- 利用 dict 函数

```
>>> aInfo = {'Wangdachui': 3000, 'Niuyun':2000, 'Linling':4500,
↳ 'Tianqi':8000}# 直接创建

>>> info = [('Wangdachui',3000), ('Niuyun',2000),
↳ ('Linling',4500), ('Tianqi',8000)] # 列表
>>> bInfo = dict(info) # 利用 dict 函数
>>> cInfo = dict(['Wangdachui',3000], ['Niuyun',2000],
↳ ['Linling',4500], ['Tianqi',8000]))
>>> dInfo = dict(Wangdachui=3000, Niuyun=2000, Linling=4500,
↳ Tianqi=8000)

>>>d=dict((( 'Wangdachui',3000),('Niuyun',2000),('Linling',4500),
↳ ('Tianqi',8000)))

# 上述创建方式均得到同样的结果:
{'Wangdachui': 3000, 'Niuyun': 2000, 'Linling': 4500, 'Tianqi':
↳ 8000}
```


例 7.11 创建员工信息表时如何将所有员工的工资默认值设置为 3000?

代码如下:

```
>>> aDict = {}.fromkeys(['Wangdachui', 'Niuyun', 'Linling',  
    ↪ 'Tianqi'], 3000)  
>>> aDict  
{'Tianqi': 3000, 'Wangdachui': 3000, 'Niuyun': 3000, 'Linling':  
    ↪ 3000}  
  
>>> sorted(aDict)  
['Linling', 'Niuyun', 'Tianqi', 'Wangdachui']
```

例 7.12 已知有姓名列表和工资列表, 如何生成字典类型的员工信息表?

代码如下:

```
>>> names = ['Wangdachui', 'Niuyun', 'Linling', 'Tianqi']  
>>> salaries = [3000, 2000, 4500, 8000]  
>>> dict(zip(names, salaries))  
{'Tianqi': 8000, 'Wangdachui': 3000, 'Niuyun': 2000, 'Linling':  
    ↪ 4500}
```

`zip()` 将对象中对应的元素打包成一个个 `tuple` (元组), 然后返回由这些 `tuples` 组成的 `list` (列表)。若传入参数的长度不等, 则返回 `list` 的长度和参数中长度最短的对象相同。

例 7.13 对于几个公司的财经数据, 如下所示, 如何构造公司代码和股票价格的字典?

```
data = [('AXP', 'American Express Company', '78.51'), ('BA', 'The Boeing  
Company', '184.76'), ('CAT', 'Caterpillar Inc.', '96.39'), ('CSCO', 'Cisco Sys-  
tems, Inc.', '33.71'), ('CVX', 'Chevron Corporation', '106.09')]
```

代码如下:

```
# 输入数据  
data = [('AXP', 'American Express Company', '78.51'),  
    ('BA', 'The Boeing Company', '184.76'),  
    ('CAT', 'Caterpillar Inc.', '96.39'),  
    ('CSCO', 'Cisco Systems, Inc.', '33.71'),  
    ('CVX', 'Chevron Corporation', '106.09')]
```

```
aList = []
bList = []
for i in range(5):
    aStr = data[i][0] # 取出数据
    bStr = data[i][2]
    aList.append(aStr)
    bList.append(bStr)
aDict = dict(zip(aList,bList))
print(aDict)
```

7.2.2 字典的操作

字典的基本操作包括：查找、更新、添加、删除、成员判断、遍历。

例 7.14 查找

```
>>> aInfo = {'Wangdachui': 3000, 'Niuyun':2000, 'Linling':4500,
↳ 'Tianqi':8000}
>>> aInfo['Niuyun']
2000
```

例 7.15 更新

```
aInfo['Niuyun'] = 9999
>>> aInfo
{'Tianqi': 8000, 'Wangdachui': 3000, 'Linling': 4500, 'Niuyun':
↳ 9999}
```

例 7.16 添加

```
>>> aInfo['Fuyun'] = 1000
>>> aInfo
{'Tianqi': 8000, 'Fuyun': 1000, 'Wangdachui': 3000, 'Linling':
↳ 4500, 'Niuyun': 9999}
```

例 7.17 删除

```
>>> del aInfo['Fuyun']
>>> aInfo
{'Tianqi': 8000, 'Wangdachui': 3000, 'Linling': 4500, 'Niuyun':
↪ 9999}
```

例 7.18 成员判断

```
>>> 'Mayun' in aInfo
False
```

例 7.19 遍历，方法如下：

```
# 遍历字典的键 key
for key in dictionaryName:
    print (key + ":" + str(dictionaryName[key]))
```

```
# 遍历字典的值 value
for value in dictionaryName.values():
    print.(value)
```

```
# 遍历字典的项
for item in dictionaryName.items():
    print.(item)
```

```
# 遍历字典的 key-value
for item, value in dictionaryName.items():
    print(item, value)
```

字典的内建函数：dict()、len()、hash()

```
>>> names = ['Wangdachui', 'Niuyun', 'Linling', 'Tianqi']
>>> salaries = [3000, 2000, 4500, 8000]
>>> aInfo = dict(zip(names, salaries))
>>> aInfo
{'Wangdachui': 3000, 'Linling': 4500, 'Niuyun': 2000, 'Tianqi':
↪ 8000}
```

```
>>> len(aInfo)
4
>>> hash('Wangdachui')
7716305958664889313
>>> testList = [1, 2, 3]
>>> hash(testList)
Traceback (most recent call last):
File "<pysHELL#127>", line 1, in <module>
hash(testList)
TypeError: unhashable type: 'list'
```

Python 还提供了丰富的字典方法：

表 7-2 常用字典方法

字典方法	含义
keys()	返回一个包含字典所有 Key 的列表
values()	返回一个包含字典所有 value 的列表
items()	返回一个包含所有键值的列表
clear()	删除字典中的所有项目
get(key)	返回字典中 key 对应的值
pop(key)	删除并返回字典中 key 对应的值
update()	将字典中的键值添加到字典中
fromkeys(iterable, value)	创建新字典，以可迭代对象中的元素分别作为字典中的键，且所有键对应同一个值
setdefault(key,[default])	如果键在字典中，返回这个键所对应的值。如果键不在字典中，向字典中插入这个键，并且以 default 为这个键的值，并返回 default。default 的默认值为 None

例 7.20 已知有员工姓名和工资信息表 {'Wangdachui':3000,'Niuyun':2000,'Linling':4500,'Tianqi':8000}，如何单独输出员工姓名和工资金额？

```
>>> aInfo = {'Wangdachui': 3000, 'Niuyun': 2000, 'Linling':
↳ 4500, 'Tianqi': 8000}
>>> aInfo.keys()# 取出字典的键 key
dict_keys(['Wangdachui', 'Niuyun', 'Linling', 'Tianqi'])
```

```
>>> aInfo.values()# 取出字典的键 key 所对应的值
dict_values([3000, 2000, 4500, 8000])
>>> for k, v in aInfo.items():
    print(k,v)

Wangdachui 3000
Niuyun 2000
Linling 4500
Tianqi 8000
```

例 7.21 人事部门有两份人员和工资信息表，第一份是原有信息，第二份是公司中有工资更改人员和新进人员的信息，如何处理可以较快地获得完整的信息表？

```
>>> aInfo = {'Wangdachui': 3000, 'Niuyun': 2000, 'Linling':
↳ 4500}
>>> bInfo = {'Wangdachui': 4000, 'Niuyun': 9999, 'Wangzi':
↳ 6000}
>>> aInfo.update(bInfo)
>>> aInfo
{'Wangzi': 6000, 'Linling': 4500, 'Wangdachui': 4000, 'Niuyun':
↳ 9999}
```

例 7.22 下面两个程序都通过键查找值，区别在哪里？你更喜欢哪一个？

```
# 通过键值直接访问
>>> stock = {'AXP': 78.51, 'BA': 184.76}
>>> stock['AAA']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'AAA'

# 通过字典的内建函数 get() 方法访问
>>> stock = {'AXP': 78.51, 'BA': 184.76}
>>> print(stock.get('AAA'))
None
```

从上述可以看出，通过键值直接访问字典，当键值不存在时会发生异常，而使用字典的内建函数 `get()` 方法则不会。

例 7.23 删除字典

```
>>> aStock = {'AXP': 78.51, 'BA': 184.76}
>>> bStock = aStock
>>> aStock = {}
>>> bStock
{'BA': 184.76, 'AXP': 78.51}

#####

>>> aStock = {'AXP': 78.51, 'BA': 184.76}
>>> bStock = aStock
>>> aStock.clear()
>>> aStock
{}
>>> bStock
{}

```

7.2.3 字典实例一

例 7.24 “统计词频”问题（统计文章其中多次出现的词语；概要分析文章内容；搜索引擎）。

统计词频 IPO 描述：

- 输入：从文件中读取一篇英文文章
- 处理：统计文件中每个单词的出现频率
- 输出：输出最常出现 10 个单词及次数图像

一种实现思路如下：

- 第一步：输入英文文章
- 第二步：建立用于词频计算的空字典
- 第三步：对文本的每一行计算词频
- 第四步：从字典中获取数据对到列表中
- 第五步：对列表中的数据对交换位置，并从大到小进行排序
- 第六步：输出结果

最后用 `Turtle` 库绘制统计词频结果图表。具体实现代码如下：

```
import turtle

## 全局变量 ##
# 词频排列显示个数
count = 10
# 单词频率数组-作为 y 轴数据
data = []
# 单词数组-作为 x 轴数据
words = []
#y 轴显示放大倍数-可以根据词频数量进行调节
yScale = 6
#x 轴显示放大倍数-可以根据 count 数量进行调节
xScale = 30

##### Turtle Start #####
# 从点 (x1,y1) 到 (x2,y2) 绘制线段
def drawLine(t, x1, y1, x2, y2):
    t.penup()
    t.goto (x1, y1)
    t.pendown()
    t.goto (x2, y2)

# 在坐标 (x,y) 处写文字
def drawText(t, x, y, text):
    t.penup()
    t.goto (x, y)
    t.pendown()
    t.write(text)

def drawGraph(t):
    # 绘制 x/y 轴线
    drawLine (t, 0, 0, 360, 0)
    drawLine (t, 0, 300, 0, 0)

    #x 轴: 坐标及描述
    for x in range(count):
```

```
x=x+1 # 向右移一位, 为了不画在源头上
drawText(t, x*xScale-4, -20, (words[x-1]))
drawText(t, x*xScale-4, data[x-1]*yScale+10, data[x-1])
drawBar(t)

# 绘制一个柱体
def drawRectangle(t, x, y):
    x = x*xScale
    y = y*yScale# 放大倍数显示
    drawLine(t, x-5, 0, x-5, y)
    drawLine(t, x-5, y, x+5, y)
    drawLine(t, x+5, y, x+5, 0)
    drawLine(t, x+5, 0, x-5, 0)

# 绘制多个柱体
def drawBar(t):
    for i in range(count):
        drawRectangle(t, i+1, data[i])
##### Turtle End #####

# 对文本的每一行计算词频的函数
def processLine(line, wordCounts):
    # 用空格替换标点符号
    line = replacePunctuations(line)
    # 从每一行获取每个词
    words = line.split()
    for word in words:
        if word in wordCounts:
            wordCounts[word] += 1
        else:
            wordCounts[word] = 1

# 空格替换标点的函数
def replacePunctuations(line):
    for ch in line:
```



```
        if ch in "~@#$$%^&*()_-=<>?/,.:;{}[]|\'\"":
            line = line.replace(ch, " ")
    return line

def main():
    # 用户输入一个文件名
    filename = input("enter a filename:").strip()
    infile = open(filename, "r")

    # 建立用于计算词频的空字典
    wordCounts = {}
    for line in infile:
        processLine(line.lower(), wordCounts)

    # 从字典中获取数据对
    pairs = list(wordCounts.items())

    # 列表中的数据对交换位置，数据对排序
    items = [[x,y] for (y,x) in pairs]
    items.sort()

    # 输出 count 个数词频结果
    for i in range(len(items)-1, len(items)-count-1, -1):
        print(items[i][1]+"\\t"+str(items[i][0]))
        data.append(items[i][0])
        words.append(items[i][1])

    infile.close()

    # 根据词频结果绘制柱状图
    turtle.title(' 词频结果柱状图')
    turtle.setup(900, 750, 0, 0)
    t = turtle.Turtle()
    t.hideturtle()
    t.width(3)
    drawGraph(t)
```

```
# 调用 main() 函数
if __name__ == '__main__':
    main()
```

运行结果如图 7-5 所示。

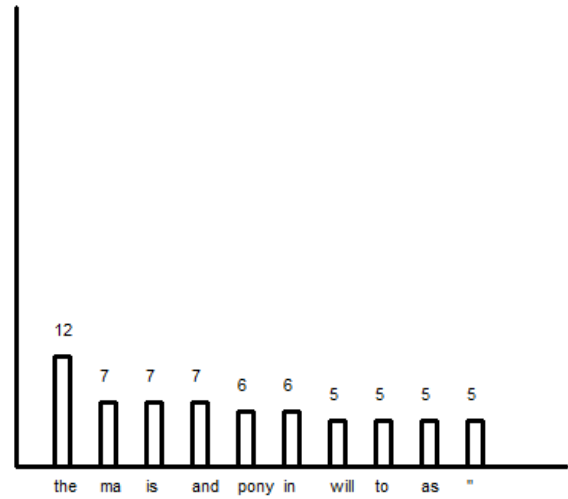


图 7-5 词频统计图

7.2.4 字典实例二

例 7.25 同前,使用字典结构将电话簿 TeleAddressBook.txt 和电子邮件 EmailAddressBook.txt 合并为一个完整的 AddressBook 文件。源数据文件格式如图 7-6 所示, 目标文件如图 7-7 所示。

TeleAddressBook.txt - 记事本					EmailAddressBook.txt - 记事本				
文件(F)	编辑(E)	格式(O)	查看(V)	帮助(H)	文件(F)	编辑(E)	格式(O)	查看(V)	帮助(H)
姓名		电话号码			姓名		邮箱		
王颐箴		13691177890			王颐箴		57320009@qq.com		
张三		13529293939			李斯		12211212@qq.com		
李斯		15811589981			王五		949495968@qq.com		
桑迪		13010026896			桑迪		wqweqe@163.com		

图 7-6 源文件数据结构

该问题的 IPO 模式可以描述为:

- 输入: 电话簿、邮箱地址文件
- 处理: 将两个文件内容进行合并

姓名	电话	邮箱
王笛颐	57320009@qq.com	13691177890
李斯	12211212@qq.com	15811589981
王五	949495968@qq.com	-----
桑迪	wqweqe@163.com	13010026896
张三	-----	13529293939

图 7-7 合并后文件数据结构

- 输出：合并后包含电话和邮箱地址簿的文件
- 程序实现的具体过程为：
- (1) 打开文件、读取文件，分别获取文件中的信息到两个列表 1 和列表 2 中
 - (2) 建立空字典 dic1，dic2 存储姓名、电话和邮箱
 - (3) 文本合并处理，具体实现方法：
 - 生成新的数据表头
 - 按字典键的操作遍历姓名列表 1，处理与表 2 重名的信息，处理其他信息
 - 处理列表 2 中剩余人的信息，合并到列表 3 中
 - (4) 将新生成的合并数据写入新文件
 - (5) 关闭所有打开文件

```
# 利用字典将两个通讯文本合并为一个文本
femail=open("EmailAddressBook.txt",'rb')
ftele=open("TeleAddressBook.txt",'rb')

femail.readline() # 跳过第一行
ftele.readline()
line_email=femail.readlines() # 读文件
line_tele=ftele.readlines()

# 字典方式保存
dict1,dict2,list_new={},{},[]

# 获取 TeleAddressBook 中的姓名、电话信息
# 'gbk' 是用来将中文写入文本，防止乱码
for line in line_tele:
    elements=line.split()
    # 将文本读出来的 bytes 转为 str 类型
    dict1[elements[0]]=str(elements[1].decode('gbk'))
```

```

# 获取 TeleAddressBook 中的姓名、邮件信息
for line in line_email:
    elements=line.split()
    dict2[elements[0]]=str(elements[1].decode('gbk'))

### 开始处理 ###
list_new.append(' 姓名\t 电话 \t 邮箱\n')

for key in dict1:
    s=''
    if key in dict2.keys():
        ↪ s='\t'.join([str(key.decode('gbk')),dict1[key],dict2[key]])
        s+='\n'
    else:
        s='\t'.join([str(key.decode('gbk')),dict1[key],str('
        ↪ ----- '))])
        s+='\n'
    list_new.append(s)

# 处理姓名列表 2 中剩余的姓名
for key in dict2:
    s=''
    if key not in dict1.keys():
        s='\t'.join([str(key.decode('gbk')),str('
        ↪ -----
        ↪ '),dict2[key]])
        s+='\n'
    list_new.append(s)

# 将合并后的输几局写入文件
with open("AddressBook.txt",'w') as f:
    f.writelines(list_new)
ftele.close()
femail.close()
f.close()
print("The addressBooks are merged!")

```

8 Python 编程之计算生态

8.1 程序设计方法

8.1.1 计算思维

计算思维（Computational Thinking）的概念由美国 CMU 计算机系主任周以真于 2006 年提出：运用计算机科学基础概念求解问题、设计系统和理解人类行为。

生活中计算思维的应用：

- 我们拥有：四个灶，锅碗瓢盆，食物原料
- 我们完成：肉菜、素菜、甜点
- 考虑因素：好吃、不能凉、搭配素菜

计算思维的人：有限资源、设定并行流程、得出最好效果：

- 输入 I：四个灶，一定数量的锅碗瓢盆，食物原料
- 处理 P：做饭过程统筹设计
- 输出 O：肉菜、素菜、甜点

计算思维的本质是理解计算特性，将计算特性抽象为计算问题，程序设计实现问题的自动求解。随计算机科学发展而提出，其特点是抽象（Abstraction）、自动化（Automation），体现实证思维、逻辑思维、计算思维。

计算机模拟解决问题：模拟现实世界计算过程提供一般情况下无法获得的信息。简单的模拟可以揭示某些困难问题的本质规律，如天气预测、飞行器设计、电影特效、核试验模拟等。

例 8.1 体育竞技分析

该问题的 IPO 模式：

- 输入 I：两个球员（A 和 B）的能力值，模拟比赛的场次
- 处理 P：模拟比赛过程
- 输出 O：球员 A 和 B 分别赢得球赛的概率

一个期望的输出结果：

- 模拟比赛数量：500
- 球员 A 获胜场次：268（53.6%）
- 球员 B 获胜场次：232（46.4%）

8.1.2 自顶向下的设计

自顶向下的设计思想如图 8-1 所示。

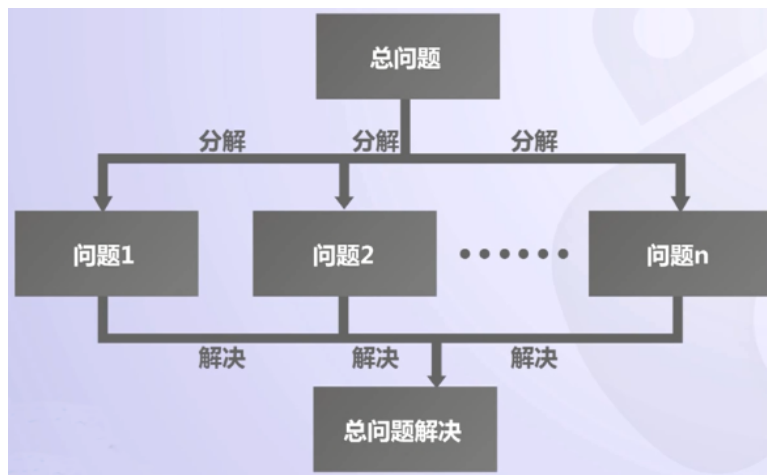


图 8-1 自顶向下设计思想

体育竞技分析程序步骤

顶层设计（第一阶段）举例

```

from random import *

def main():
    printIntro() # 打印程序的介绍信息
    probA, probB, n = getInputs() # 获取程序运行所需的参数: ProbA、
    ↪ ProbB、n
    winsA, winsB = simNGames(n, probA, probB) # 模拟 n 次比赛
    PrintSummary(winsA, winsB) # 输出球员 A 和 B 获胜比赛的次数
    ↪ 和概率
  
```

体育竞技分析程序第一阶段的结构如图 8-2 所示。

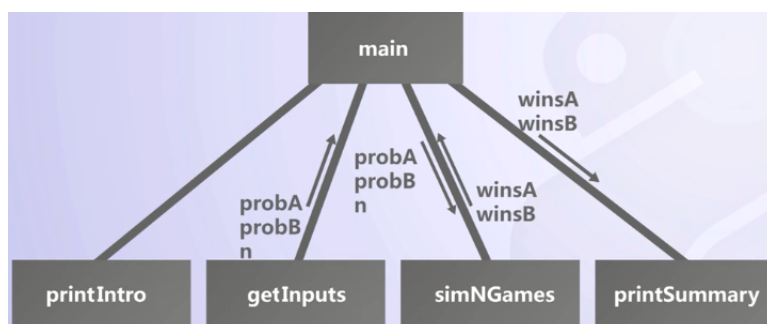


图 8-2 体育竞技分析程序结构图：第一阶段

体育竞技分析程序第二阶段的结构如图 8-3 所示。

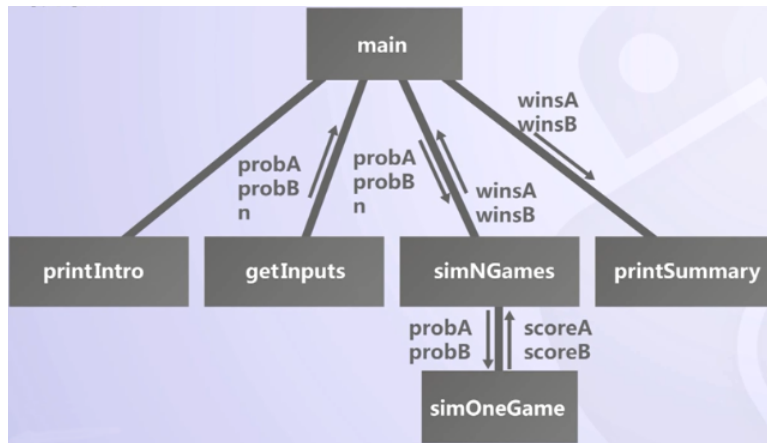


图 8-3 体育竞技分析程序结构图：第二阶段

```

# printIntro() 函数打印体育竞技程序基本信息
def printIntro():
    print('This program simulates a game between two')
    print('There are two players, A and B')
    print('Probability(a number between 0 and 1)is used')

# getInputs() 函数获得 A、B 获胜概率以及需要模拟的比赛次数
def getInputs():
    a = eval(input('What is the prob.player A wins?'))
    b = eval(input('What is the prob.player B wins?'))
    n = eval(input('How many games to simulate?'))
    return a,b,n

# simNGames() 函数（核心）模拟 n 场比赛，并记录每个球员得分
def simNGames(n,probA,probB):
    winsA = 0
    winsB = 0
    for i in range(n):
        scoreA,scoreB = simOneGame(probA,probB)
        if scoreA >scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA,winsB
  
```

体育竞技分析程序第三阶段的结构如图 8-4 所示。

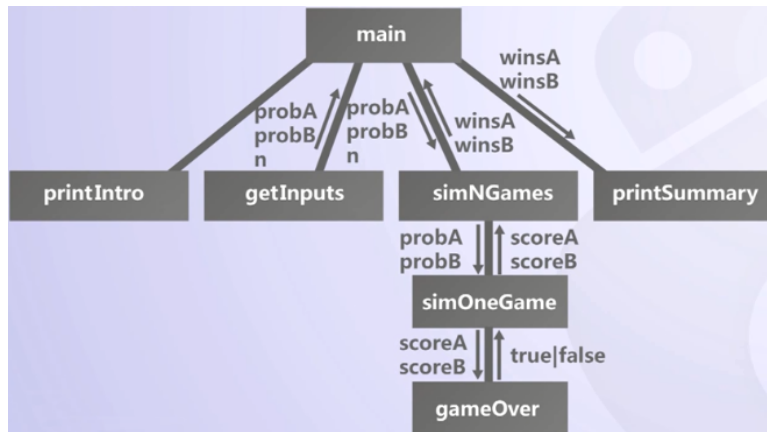


图 8-4 体育竞技分析程序结构图：第三阶段

```

# 一场比赛模拟
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while not gameOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB

# 判断比赛是否结束
def gameOver(a, b):
    return a==15 or b==15

# 打印比赛结果
def PrintSummary(winsA, winsB):

```



```
n = winsA + winsB
print('\nGames simulated:%d'%n)
print('Wins for A:{0}({1:0.1%})'.format(winsA,winsA/n))
print('Wins for B:{0}({1:0.1%})'.format(winsB,winsB/n))
```

上述完整代码如下：

```
from random import *

def main():
    printIntro() # 打印程序的介绍信息
    probA,probB,n = getInputs() # 获取程序运行所需的参数：ProA、
    ↪ ProB、n
    winsA, winsB = simNGames(n,probA,probB) # 模拟 n 次比赛
    PrintSummary(winsA, winsB) # 输出球员 A 和 B 获胜比赛的次数
    ↪ 和概率

def printIntro():
    print('This program simulates a game between two')
    print('There are two players, A and B')
    print('Probability(a number between 0 and 1)is used')

def getInputs():
    a = eval(input('What is the prob.player A wins?'))
    b = eval(input('What is the prob.player B wins?'))
    n = eval(input('How many games to simulate?'))
    return a,b,n

def simNGames(n,probA,probB):
    winsA = 0
    winsB = 0
    for i in range(n):
        scoreA,scoreB = simOneGame(probA,probB)
        if scoreA >scoreB:
            winsA = winsA + 1
        else:
```

```
        winsB = winsB + 1
    return winsA,winsB
def simOneGame(probA,probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while not gameOver(scoreA,scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA,scoreB

def gameOver(a,b):
    return a==15 or b==15

def PrintSummary(winsA, winsB):
    n = winsA + winsB
    print('\nGames simulated:%d'%n)
    print('Wins for A:{0}({1:0.1%})'.format(winsA,winsA/n))
    print('Wins for B:{0}({1:0.1%})'.format(winsB,winsB/n))

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

自顶向下设计过程总结：

- 将算法表达为一系列小问题
- 为每个小问题设计接口
- 通过将算法表达为接口关联的多个小问题来细化算法
- 为每个小问题重复上述过程

8.1.3 自底向上的执行

自顶向下的设计就是从顶层开始分解问题为更小的问题进行求解。

自底向上的执行就是从底层模块开始一个一个进行测试，程序写好后，需要通过运行程序进行测试。

软件测试

- 小规模程序
 - 直接运行
- 中等规模
 - 从就结构图底层开始，逐步上升
 - 先运行每个基本函数，再测试整体函数
- 较大规模
 - 高级软件测试方法

例 8.2 体育竞技分析框架

```
#matchSim.py
from random import random
def main():
def printIntro():
def getInput():
def simNGames():
def simOneGame()
def gameOver():
If __name__ == '__main__':main()
```

观察体育竞技分析程序第三阶段的结构图 8-5。

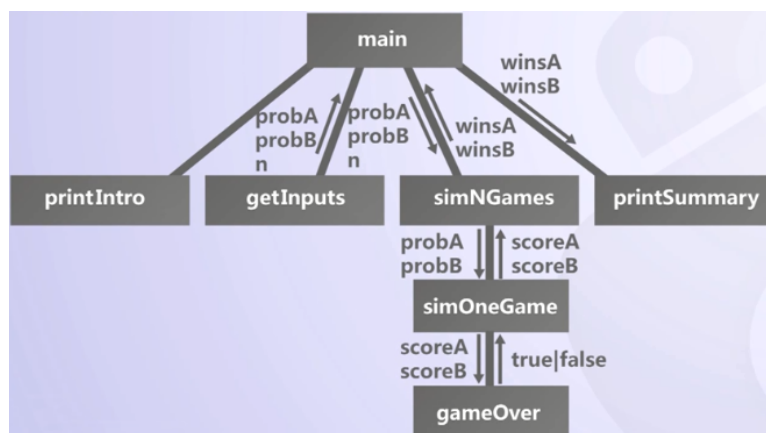


图 8-5 体育竞技分析程序结构图：第三阶段

我们可以从最底层的“gameOver()”函数开始进行测试。这里可以采用调用库函数的方法，即通过 `import` 将 `matchSim` 文件加载到交互环境，并通过 `function_name` 开展测试，如下所示：

```
# 单元测试
>>> import matchSim
>>> matchSim.gameOver(0,0)
False
>>> matchSim.gameOver(5,10)
False
>>> matchSim.gameOver(15,3)
True
>>> matchSim.gameOver(3,15)
True
```

`simOneGame()` 函数具有概率属性行为，因此无法确定输出是什么，测试其最好方法是理解其行为的合理性，例如两个球员的获胜概率均为 0.5，模拟一次比赛，其比分为 15: 13，再次模拟，其比分则变为 12: 15。注意到当二者概率相等时，比分也十分接近；当两者获胜概率相差较大时，则比赛结果呈现压倒性趋势，这与函数预期功能是相符的。

```
# 单元测试
>>> matchSim.simOneGame(0.5,0.5)
(15, 13)
>>> matchSim.simOneGame(0.5,0.5)
(12, 15)
>>> matchSim.simOneGame(0.3,0.3)
(15, 13)
>>> matchSim.simOneGame(0.3,0.3)
(12, 15)

>>> matchSim.simOneGame(0.4,0.9)
(1, 15)
>>> matchSim.simOneGame(0.4,0.9)
(4, 15)
>>> matchSim.simOneGame(0.9,0.4)
(15, 3)
>>> matchSim.simOneGame(0.9,0.4)
```

(15, 0)

通过继续进行单元测试，可以检测程序中的每个函数。独立检验每个函数更容易发现错误，当把整个程序检测一遍后，程序运行将更为顺利。

最后回到体育竞技分析问题上，通过模拟方法分析两球员之间微小能力差距所带来的结果差异：是否会出现微小能力差距导致比赛结果一边倒的现象？程序分析代码如下：

```
What is the prob.player A wins?0.65
```

```
What is the prob.player B wins?0.6
```

```
How many games to simulate?5000
```

```
Games simulated:5000
```

```
Wins for A:3320(66.4%)
```

```
Wins for B:1680(33.6%)
```

通过上述模拟结果，我们可以知道：尽管两球员能力差距微小，仅为 0.05，球员 A 大约需要经历 3 场比赛才能获得一次胜利，可见 A 球员进行一场 3 局或 5 局的比赛取得胜利的机会比较渺茫。

进一步，可将该程序拓展为羽毛球、乒乓球等多种模式，就有可能找到不同比赛规则下的竞技规律。

8.2 软件开发方法基础

8.2.1 软件开发方法

软件是能够完成预定功能和性能的可执行的计算机程序、支持程序正常运行的数据、以及描述程序的操作和使用的文档的集合。

软件工程，将系统的、严格约束的、可量化的方法应用于软件的开发、运行和维护。简单来说，就是将工程化应用于软件。

软件开发生命周期：确定问题、可行性分析、系统分析、系统设计、编码、测试、安装和维护。如图 8-6 所示。

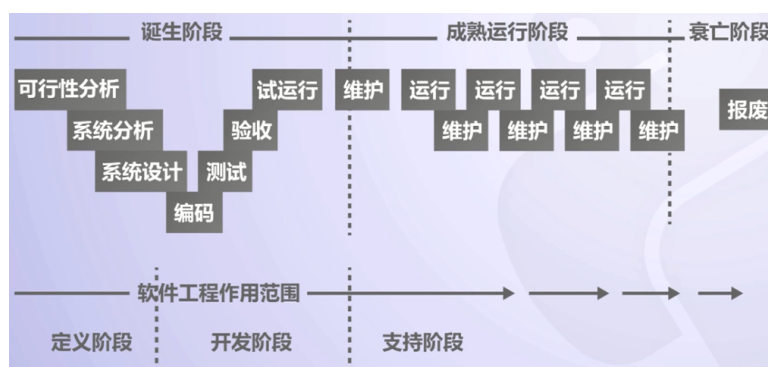


图 8-6 软件开发生命周期

软件开发模式

- 瀑布模式：重视各个阶段的顺序性，当一个阶段的文档获得认可才进入下一阶段，如图 8-7 所示。

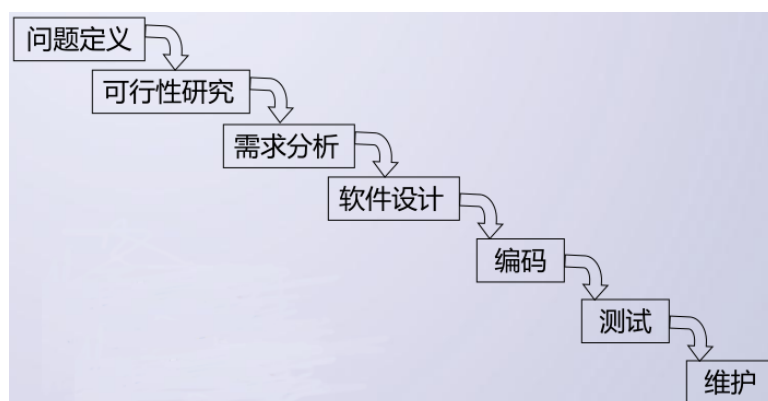


图 8-7 瀑布模式

- 螺旋模式：设计、执行并测试原型；再设计、执行并测试新特征；将原型逐步扩展为最终程序。如图 8-8 所示。
- 快速原型模型

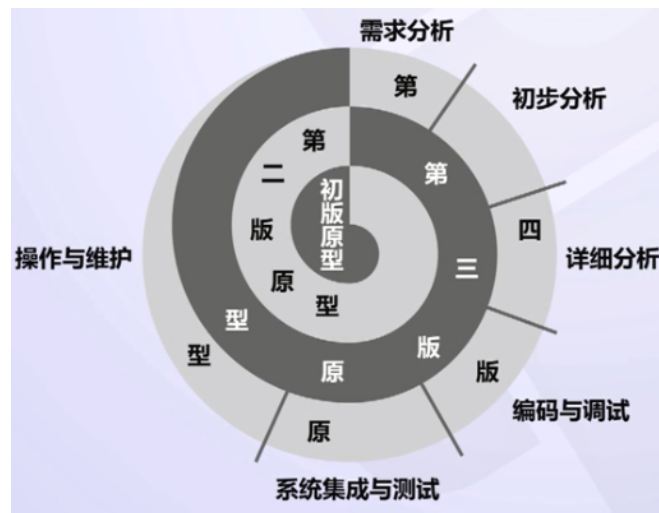


图 8-8 螺旋模式

- 喷泉模式
- 混合模式
- 敏捷开发模式

体育竞技分析

本质：模拟一场比赛 `simOneGame()`。

原型：

- 假设每个球员都有机会在 50 对 50 的概率下赢得一分
- 打了 30 回合
- 谁会得分或改变球权

原型的例子：

```
import random
def simOneGame():
    scoreA = 0
    scoreB = 0
    serving = "A"
    for i in range(30):
        if serving == "A":
            if random() < 0.5:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < 0.5:
```

```
        scoreB = scoreB + 1
    else:
        serving = "A"
    print(scoreA)
    print(scoreB)

simOneGame()
```

阶段 1: 构建最初的原型

阶段 2: 添加两个参数代表两个球员赢球的概率

阶段 3: 进行比赛, 直到一个球员达到 15 分

阶段 4: 将比赛扩展为多场比赛

阶段 5: 建立完整的程序

8.2.2 敏捷开发方法

敏捷开发

- 以人为核心、迭代、循序渐进
- 针对传统的瀑布开发模式的弊端
- 分为多个相互联系、独立运行的小项目
- 软件一直处于可使用状态

敏捷开发更符合软件开发的规律: 自底向上、逐步有序、遵循软件客观规律、迭代增量开发。

敏捷开发效率更高。在传统方式中, 管理者“控制”团队; 团队成员被动的等待指令、独立工作、协作少; 在敏捷开发方式中, 管理者“激发”团队; 团队成员共同参与。

敏捷开发采用轻量级软件开发方法。常见的开发方法:

- Scrum 是一种流行的敏捷开发框架, 由一个开发过程、几种角色、一套规范的实施方法组成。
- 极限编程 (XP)
- 精益开发 (Lean Development)
- 动态系统开发方法 (DSDM)
- 特征驱动开发 (Feature Driver development)
- 水晶开发 (Cristal Clear)

敏捷开发典型过程:

(1) 对产品形成共识

(2) 建立和维护产品需求列表, 并进行优先级排序

- (3) 筛选高优先级需求进入本轮迭代开发
- (4) 细化本轮迭代需求，一次在本轮迭代完成
- (5) 每日召开站立会议（任务看板：任务未完成、正在做、已完成）
- (6) 对每轮迭代交付的可工作软件，进行现场验收和反馈
- (7) 从第 (3) 步开始，开始下一轮迭代

8.3 面向过程程序设计

面向过程的程序设计采用以程序执行过程为设计流程的思想，是程序设计中最自然的一种设计方法。面向过程的程序设计也叫结构化编程。

例 8.3 铅球飞行计算问题。如图 8-9 所示，在给定不同的投掷角度和初始速度下，求解计算铅球的飞行距离。

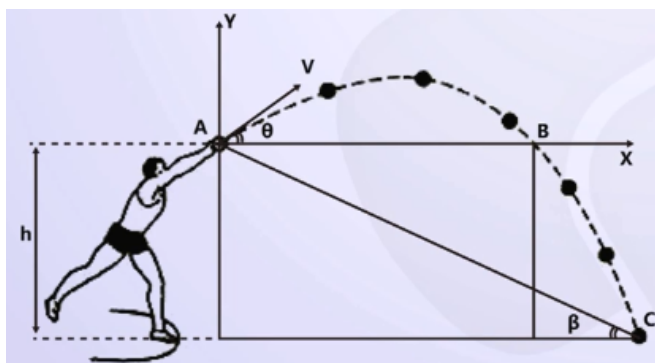


图 8-9 铅球飞行示意图

IPO 描述为：

- 输入：铅球发射角度、初始速度 (m/s)、初始高度 (m)
- 处理：模拟铅球飞行，时刻更新铅球在飞行中的位置
- 输出：铅球飞行距离 (m)

简化问题：

- 忽略空气阻力
- 重力加速度 9.8 m/s^2
- 铅球飞行过程：铅球高度、飞行距离
- 时刻更新铅球在飞行中的位置：
 - 假设起始位置是点 (0,0)
 - 垂直方向上运动距离 (y 轴)
 - 水平方向上移动距离 (x 轴)

设计参数：

- 仿真参数：投掷角度 `angle`、初始速度 `velocity`、初始高度 `height`、飞行距离 `interval`
- 位置参数：x 轴坐标 `xpos`，y 轴坐标 `ypos`
- 速度分量：x 轴方向上速度 `xvel`，y 轴方向上速度 `yvel`

代码如下：

```
from math import pi, sin, cos, radians

def main():
    # 根据提示输入仿真参数：投掷角度 angle、初始速度 velocity、初
    ↪ 始高度 height、飞行距离 interval
    angle = eval(input("Enter the launch angle (in degrees):"))
    vel = eval(input("Enter the initial velocity (in
    ↪ meters/sec):"))
    h0 = eval(input("Enter the initial height (in meters):"))
    time = eval(input("Enter the time interval: "))

    xpos, ypos = 0, h0
    theta = radians(angle) # 角度转化为弧度
    xvel = vel * cos(theta) # 计算初始速度分量
    yvel = vel * sin(theta)

    while ypos >= 0: # 计算 x 轴坐标 xpos、速度 xvel, y 轴坐标
        ↪ ypos、速度 yvel
        xpos = xpos + time * xvel
        yvell = yvel - time * 9.8
        ypos = ypos + time * (yvel + yvell)/2.0
        yvel = yvell
        print("\nDistance traveled:{0:0.1f}meters.".format(xpos))

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

程序模块化如下：

```
from math import pi, sin, cos, radians

#getInputs() 函数获得仿真参数：投掷角度 angle、初始速度 velocity、
↪ 初始高度 height、飞行距离 interval
```

```
def getInputs():
    angle = eval(input("Enter the launch angle (in degrees):"))
    vel = eval(input("Enter the initial velocity (in
        ↪ meters/sec):"))
    h0 = eval(input("Enter the initial height (in meters):"))
    time = eval(input("Enter the time interval: "))
    return angle,vel,h0,time

#getXComponents(vel,angle) 获取 x、y 方向上的速度分量
def getXComponents(vel,angle):
    theta = radians(angle) # 角度转化为弧度
    xvel = vel * cos(theta) # 计算初始速度分量
    yvel = vel * sin(theta)
    return xvel,yvel

#updatePosition(time,xpos,ypos,xvel,yvel) 函数更新速度分量并计算
    ↪ 飞行距离
def updatePosition(time,xpos,ypos,xvel,yvel):
    xpos = xpos + time * xvel
    yvell = yvel - time * 9.8
    ypos = ypos + time * (yvel + yvell)/2.0
    yvel = yvell
    return xpos,ypos,yvel

def main():
    angle,vel,h0,time=getInputs()
    xpos,ypos =0,h0
    xvel,yvel=getXYComponents(vel,angle)
    while ypos >= 0:
        xpos,ypos,yvel=updatePosition(time,xpos,ypos,xvel,yvel)
    print("\nDistance traveled:{0:0.1f}meters.".format(xpos))

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

面向过程程序设计基本步骤:

- 分析程序从输入到输出的各步骤
- 按照执行过程从前到后编写程序
- 将高耦合部分封装成模块或函数
- 输入参数，按照程序执行过程调试

总结面向过程程序设计特点：

- 通过分步骤、模块化：将一个大问题分解成小问题；将一个全局过程分解为一系列局部过程
- 面向过程：最为自然、也是最贴近程序执行过程的程序设计思想；在面向对象的程序设计中也会使用面向过程的设计方法

8.4 面向对象程序设计

真实世界的对象可以用状态和行为两个特征来描述。比如：猫，状态：名字、颜色、品种；行为：喵叫、摇尾巴、捉老鼠。

类：某种类型集合的描述。例如，人：

- 属性：类本身的一些特性，如名字、身高和体重等属性，属性具体值则会根据每个人的不同而不同
- 方法：类所能实现的行为，如吃饭、走路和睡觉等方法

类的定义

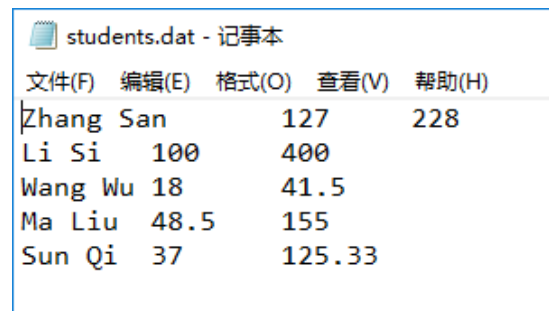
```
class classname[(父类名)]:  
    [成员函数及成员变量]
```

- `_init_` 构造函数：初始化对象的各属性
- `_del_` 析构函数：销毁对象

例 8.4 学生课程评估：学分和平均绩点 GPA。绩点计算以 GPA 4 分为准则，例如一门课程 3 学分，某位同学得了“A”，则量分数为 $3 \times 4 = 12$ 。记录学生成绩文件 `students.dat` 的数据结构如图 8-10 所示。编写程序，通过读取文件找出平均绩点最高的学生，然后输出学生的名字、学分和平均绩点。

GPA 算法描述的伪代码如下：

```
获取文件名  
打开文件  
设置第一个学生为 best  
对文件中的每一个学生  
    if s.gpa() > best.gpa()
```



Zhang San	127	228
Li Si	100	400
Wang Wu	18	41.5
Ma Liu	48.5	155
Sun Qi	37	125.33

图 8-10 学生成绩文件 students.dat 的数据结构

设置 `s` 为 `best`
打印 `best` 学生的信息

具体实现代码如下：

找到 GPA 最高的学生

`class Student:` # 定义 `Student` 类

`def __init__(self, name, hours, qpoints):`

`self.name = name` # 初始化变量

`self.hours = float(hours)`

`self.qpoints = float(qpoints)`

`def getName(self):`

`return self.name`

`def getHours(self):`

`return self.hours`

`def getQPoints(self):`

`return self.qpoints`

`def gpa(self):`

`return self.qpoints/self.hours`

`def makeStudent(infoStr):`

`name, hours, qpoints = infoStr.split("\t")`

`return Student(name, hours, qpoints)`

```
def main():
    # 打开输入文件
    filename = input("Enter name the grade file: ")
    infile = open(filename, 'r')
    # 设置文件中第一个学生的记录为 best
    best = makeStudent(infile.readline())

    # 处理文件剩余行数据
    for line in infile:
        # 将每一行数据转换为一个记录
        s = makeStudent(line)
        # 如果该学生是目前 GPA 最高的，则记录下来
        if s.gpa() > best.gpa():
            best = s
    infile.close()

    # 打印 GPA 成绩最高的学生信息
    print("The best student is:", best.getName())
    print("hours:", best.getHours())
    print("GPA:", best.gpa())

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

运行结果如下：

```
Enter name the grade file: students.dat
The best student is: Sun Qi
hours: 37.0
GPA: 3.3872972972972972
```

面向对象的程序设计的基本步骤：

第一步：根据功能，轴向业务对象。

第二步：构建独立的业务模块，利用封装、继承、多态等抽象业务需求。

第三步：编写程序。

第四步：以对象为单位输入参数、开展测试。

8.5 面向对象实例

采用面向对象的程序设计理念考虑例 8.3 铅球飞行轨迹计算。

- 铅球对象属性: xpos、ypos、xvel、yvel
- 构建投射体类 Projectile
- 创建和更新对象的变量

投射体类 Projectile 代码如下:

```

1  from math import sin, cos, radians
2
3  class Projectile:
4      # 包括一个构造函数和三个方法
5      def __init__(self, angle, velocity, height):
6          # 根据给定的发射角度、初始速度和位置创建一个投射体对象
7          self.xpos = 0.0
8          self.ypos = height
9          theta = radians(angle)
10         self.xvel = velocity * cos(theta)
11         self.yvel = velocity * sin(theta)
12
13     def update(self, time):
14         # 更新投射体的状态
15         self.xpos = self.xpos + time * self.xvel
16         yvell = self.yvel - 9.8 * time
17         self.ypos = self.ypos + time * (self.yvel + yvell) / 2.0
18         self.yvel = yvell
19
20     def getY(self):
21         # 返回投射体的角度
22         return self.ypos
23
24     def getX(self):
25         # 返回投射体的距离
26         return self.xpos

```

引入对象，程序模块化

```
from Projectile import *
```

```
#getInputs() 函数获得仿真参数: 投掷角度 angle、初始速度 velocity、
↪ 初始高度 height、飞行距离 interval
```

```
def getInputs():
    angle = eval(input("Enter the launch angle (in degrees):"))
    vel = eval(input("Enter the initial velocity (in
        ↪ meters/sec):"))
    h0 = eval(input("Enter the initial height (in meters):"))
    time = eval(input("Enter the time interval: "))
    return angle, vel, h0, time

def main():
    angle, vel, h0, time = getInputs()
    shot = Projectile(vel, angle, h0)
    while shot.getY() >= 0:
        shot.update(time)
    print("\nDistance
        ↪ traveled:{0:0.1f}meters.".format(shot.getX()))

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

8.6 面向对象的特点

面向对象的特点：封装、多态和继承。

8.6.1 封装

封装：从业务逻辑中抽象对象时，赋予对象相关数据与操作，把一些数据和操作打包在一起的过程就是封装。

对象的实现和使用是独立的

支持代码复用

比如例 8.5 中的 Projectile 将投射体属性和方法封装在类的内部，不必关心铅球内部如何实现。Projectile 类可以被多个程序、多个对象所使用。

8.6.2 多态

多态

- 对象怎么回应一个依赖于对象类型或种类的消息

- 在不同情况下用一个函数名启用不同方法
- 灵活性

一个图形对象列表: Circle、Rectangle、Polygon, 使用相同代码可以画出列表中所有的图形:

```
for obj in objects:  
    obj.draw(win)
```

8.6.3 继承

继承: 一个类 (subclass) 可以借用另一个类 (superclass) 的行为。

- 避免重复操作
- 提升代码复用程度

例如员工信息系统:

- Employee 类 # 包含所有员工通用一般信息
 - Employee 类属性 homeAddress() # 返回员工住址信息
- Employee 子类: SalariedEmployee 和 HourlyEmployee
 - 共享 homeAddress() 属性
 - 自己 monthlyPay() 属性

参考文献

- [1] 嵩天, 黄天羽. 零基础学 python 语言 [J/OL]. 中国大学 MOOC. <http://www.icourse163.org/course/BIT-1002058035>.