第3章

Python 语言初步

3.1 Python 语言概述

3.1.1 Python 语言简介

什么是 Python? Python 是一种灵活多用的计算机程序设计语言，使用 Python 进行的编程语法特色更强，具有更高的可读性。Python 对于初级程序员来说非常友好，语法简单易懂，应用广泛，实用性强。**Python 是一种解释型语言**，解释型语言指的是源代码先被翻译成中间代码，再由解释器对中间代码进行解释运行，这就意味着 Python 的跨平台性很好，所有支持 Python 语言的解释器都可以运行 Python。**Python 是交互式语言**，它可以在交互界面直接执行代码，大多数 Linux 系统都使用 Python 语言作为基本配置。Python **是面向对象语言**，这意味着 Python 支持面向对象的风格或代码封装在对象的编程技术。

1. Python 的发展历史

Python 由荷兰人 Guido van Rossum 于 1989 年创造，并于 1991 年发布第一个公开发行版。自 2004 年以后，Python 的使用率大幅增长，Python 2 于 2000 年 10 月发布，稳定版本是 Python 2.7。Python 3 于 2008 年 12 月发布，不完全兼容 Python 2。

2. Python 的特点

Python 语言**简单易学**，有较为简单的语法，也很容易入门与上手操作。**免费、开源**，Python 的一个很重要的优点是它是免费开源的，开发者可以自由地发布这个软件的复制版、阅读它的源代码、对它加以改动、取其部分用于新的自由软件中。**可移植性**，由于它的开源本质，Python 有很强的可移植性能，如果避免使用了依赖于系统的特性，那么所有 Python 程序无须修改就可以在多个平台上面运行。

3. 为什么使用 Python

Python 提供了很多的重要库，包括 NumPy、Pandas、Matplotlib、SciPy、

scikit-learn 等。这些库为 Python 提供了数值计算需要的多种数据结构、算法、接口及多种机器学习库函数，以及用于制图和可视化的必备操作等。这些功能都使得 Python 用起来更为方便，也因此成为了解决数据挖掘、数据分析和机器学习问题的必备工具。同时 Python 编写小型程序、脚本也十分方便，能够快速处理各种类型的数据，在服务器命令行环境上运行简单明了。因此，综合考虑 Python 的各个方面的优点，我们选择使用 Python 作为本书分析与挖掘数据的首要语言。

3.1.2　Python 语言环境搭建

1. Python 的下载与安装

Python 的版本有 2.x 和 3.x 两大类，其中比较常用的是 Python 2.7 和 Python 3.6。由于 Python 不支持向下兼容，在 Python 3.x 的环境下，Python 2.x 版本的代码不一定能正常运行。因此，在使用 Python 进行开发与移植时，需要了解对于 Python 版本的需求。

另一方面，Python 的使用通常需要与多种附加包结合，而且应用场景也多种多样。因此，需要附加的安装操作也多种多样，没有统一的解决方案。在本书中推荐使用免费的且集成较多安装包的 Anaconda 软件来安装 Python 3.6 以及相应的环境。

下面分别对 3 个系统上的 Anaconda 安装过程进行简述。

Windows：先从(http://anaconda.com/downloads)下载 Anaconda 安装器，并按照官网下载页上的安装说明进行安装。安装结束后需要确定所有的设置是否都正确，确认方法如下：打开命令行应用(cmd.exe)，输入 Python 来启动 Python 解释器，如果能够正确输出符合你下载的 Anaconda 版本的信息，则证明安装成功。

```
C:\Windows\system32>Python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit
(AMD64)] on win32
>>>
```

Mac OS：下载 OS X 版的 Anaconda 安装器，命令如 Anaconda3-4.1.0-MacOSX-x86_64.pkg。双击.pkg 文件运行安装器。安装器运行时，会自动将 Anaconda 执行路径添加到.bash_profile 文件中，该文件位于/Users/$USER/.bash_profile。确认安装正常，可以尝试在系统命令行(打开终端应用得到命令提示符)中运行 IPython：

```
$iPython
```

要退出命令行，按下 Ctrl+D 键，或输入命令 exit()后按回车。

Linux：Linux 下的安装细节取决于所用的 Linux 版本，安装器是一段 Shell 脚本，安装时需要使用一个文件名类似于 Anaconda3-4.1.0-Linux-x86_64.sh 的文件，并在 bash 命令行中输入：

```
$bash Anaconda3-4.1.0-Linux-x86_64.sh
```

在接受许可后，通常选用默认安装设置就可以成功安装。

2. 集成开发环境

Python 开发环境的基准是“IPython 加文本编辑器”。通常会在 IPython 或 Jupyter notebook 中写一段代码，然后迭代测试、调试。这种方法有助于在交互情况下操作数据，并可以通过肉眼确认特定数据集是否做了正确的事。

然而，当开发软件时，使用者可能倾向于使用功能更为丰富的集成开发环境（IDE）而不是功能相对简单的文本编辑器。常用的 IDE 如下。

（1）PyDev（免费），基于 Eclipse 平台的 IDE。

（2）PyCharm，Jetbrains 公司开发，对商业用户收费，对开源用户免费。

（3）Spyder，Anaconda 集成的 IDE。

（4）Python Tools for Visual Studio，适合 Windows 用户。

具体的选择，则需要读者根据自己的需求进行自行判断。在本书中，使用 Anaconda 自带的 Spyder 和 Jupyter notebook 进行编程。

3.2 Python 的基本用法

3.2.1 列表与元组

1. 序列概述、常用的序列操作

在 Python 中，最常用且最基础的数据结构是序列，英文为 sequence。序列中的每个元素都设有对应的编号，并且使用这些标号来查找这些元素，因此也通常称编号为这些元素的位置或索引。另外一点非常重要的是，序列的索引从 0 开始，即第一个元素的索引为 0，第二个元素的索引为 1，以此类推。

Python 中有多种序列，但最常见的是列表和元组两类，它们除了最基本的操作相同外，也会有一定的区别：列表中的元素是可以修改的，而元组不可以修改。

适用于所有序列的操作包括索引、切片、相加、相乘和成员资格检查等，下面通过举出一些例子来对这些内容进行详细讲解。

1）索引

使用索引来查找序列中的元素，索引的描述方式为“[]”。方框中的元素表示索引的元素位置，它可以取正数，也可以取负数。当使用负数 $-n$ 时，表示从右（即从最后一个元素）开始往左数，查找第 n 个元素。

【例 3-1】

```
In[1]:   number_list = [0,1,2,3,4,5,6,7,8,9]
In[2]:   number_list[3]
Out[2]:  3
In[3]:   number_list[-1]
Out[3]:  9
```

2）切片

索引通常用来访问单个元素，而当想要访问多个元素时，通常采用切片（Slicing）的方式。使用方括号截取特定范围内的元素，这种操作就是切片。切片本质上是被冒号间隔的两个索引，用来截取从第一个索引下标到第二个索引下标之间的元素。其中第一个索引指定的元素包含在切片内，但第二个索引指定的元素不包含在切片内。

【例 3-2】

```
In[4]:  number_list = [0,1,2,3,4,5,6,7,8,9]
In[5]:  number_list[3:7]
Out[5]: [3, 4, 5, 6]
```

同样，当要从列表末尾开始访问元素时，可以使用负数索引。

【例 3-3】

```
In[6]:  number_list[-3:-1]
Out[6]: [7, 8]
```

省略第一个索引时表示切片开始于序列开头，省略第二个索引时表示切片结束于序列末尾，而当两个索引都省略时，则选取整个序列。

【例 3-4】

```
In[7]:  number_list[-5:]
Out[7]: [5, 6, 7, 8, 9]
In[8]:  number_list[:5]
Out[8]: [0, 1, 2, 3, 4]
```

除此之外，在索引 1 和索引 2 之间可以设置步长来指定访问时跳跃的幅度。步长为正表示从前向后访问，步长为负表示从后向前访问。

【例 3-5】

```
In[9]:  number_list[1:7:2]
Out[9]: [1, 3, 5]
In[10]: number_list[8:2:-2]
Out[10]:[8, 6, 4]
```

3）序列相加

可以使用加法运算符“＋”来拼接序列。

【例 3-6】

```
In[11]:  list1=[1,2,3]
         list2=[4,5,6]
In[12]:  list1+list2
Out[12]: [1, 2, 3, 4, 5, 6]
```

4）乘法

在序列中，乘法用 * 表示，当序列与数 x 相乘时，将重复这个序列 x 次来创建一个新序列。

【例 3-7】

```
In[13]:  [1,2] * 8
Out[13]: [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

5）成员资格检查

通过使用 in 运算符来检查特定的值是否包含在序列中。in 用来检查满足某条件的元素是否存在，如果存在，返回 True；不存在，则返回 False。

【例 3-8】

```
In[14]:  name_list = ['Tom','Jerry','Mickey','Mike']
In[15]:  'Tom' in name_list
Out[15]: True
In[16]:  'Marry' in name_list
Out[16]: False
```

2. 列表概述、常用的列表操作

列表是序列的一种，可以对列表执行所有的序列操作，但是不同之处在于列表是可以修改的。因此，接下来会介绍一些修改列表的方法：给元素赋值、删除元素、给切片赋值以及使用列表的方法。

1）列表的修改操作

（1）给元素赋值：给元素赋值之前，需要使用索引法找到特定位置的元素，然后使用赋值号“＝”给元素赋值。但需要注意的是，不能对超出列表长度范围的元素进行赋值。

【例 3-9】

```
In[17]:  lst = [1,10,20,30,40]
In[18]:  lst[3]=35
In[19]:  lst
Out[19]: [1, 10, 20, 35, 40]
In[20]:  lst[5]=50
Out[20]: IndexError: list assignment index out of range
```

（2）删除元素：使用下标索引并结合 del 语句来删除元素。

【例 3-10】

```
In[21]:  lst = [1,10,20,30,40]
In[22]:  del lst[3]
In[23]:  lst
Out[23]: [1, 10, 20, 40]
```

(3) 使用切片可以同时对多个元素进行赋值,甚至可以实现序列的长度改变。不仅如此,使用切片赋值还可以在不替换原有元素的情况下更新元素。

【例 3-11】

```
In[24]:  number_list = [1,2,3,4,5,6]
In[25]:  number_list[2:]=[20,30,40]
In[26]:  number_list
Out[26]: [1, 2, 20, 30, 40]
In[27]:  number_list[1:1]=[5,10,15]
In[28]:  number_list
Out[28]: [1, 5, 10, 15, 2, 20, 30, 40]
In[29]:  number_list[1:4] = []
In[30]:  number_list
Out[30]  [1, 2, 20, 30, 40]
```

2) 列表方法

首先,我们要了解什么是方法,所谓方法(Method)通常会加上对象(Object)和句点来调用:object.method(arguments),方法用来实现某些功能。列表中包含一些常用的方法,可以用来查看或修改内容,下面则给出一些常用的例子来说明。

(1) **append 方法**:使用 append()函数,将某一个对象附加到列表的末尾。

【例 3-12】

```
In[31]:  number_list.append(50)
In[32]:  number_list
Out[32]: [1, 2, 20, 30, 40, 50]
```

(2) **clear 方法**:使用 clear()函数来清空列表中的所有内容。

【例 3-13】

```
In[33]:  number_list.clear()
In[34]:  number_list
Out[34]: []
```

(3) **copy 方法**:复制列表。

【例 3-14】

```
In[35]:  number_list = [1,2,3,4,5,6,7]
In[36]:  nl = number_list.copy()
In[37]:  number_list[3] = 40
In[38]:  number_list
Out[38]: [1, 2, 3, 40, 5, 6, 7]
In[39]:  nl
Out[39]: [1, 2, 3, 4, 5, 6, 7]
```

（4）**insert 方法**：将一个对象插入到列表中。

【例 3-15】

```
In[39]:  number_list.insert(2,7)
In[40]:  number_list
Out[40]: [1, 2, 7, 3, 40, 5, 6, 7]
```

（5）**remove 方法**：用于删除第一个为指定值的元素。

【例 3-16】

```
In[41]:  sentence = ['how','do','you','do']
In[42]:  sentence.remove('do')
In[43]:  sentence
Out[43]: ['how', 'you', 'do']
```

（6）**count 方法**：统计某个元素在列表中出现的次数。

【例 3-17】

```
In[44]:  ['to', 'be', 'or', 'not', 'to', 'be'].count('to')
Out[44]: 2
```

（7）**extend 方法**：在列表末尾一次性追加另一个序列中的多个值。

【例 3-18】

```
In[45]:  a1 = ['a','b','c']
         b1 = ['d','e','f']
In[46]:  a1.extend(b1)
In[47]:  a1
Out[47]: ['a', 'b', 'c', 'd', 'e', 'f']
```

（8）**index 方法**：在列表中查找指定值第一次出现的索引。

【例 3-19】

```
In[48]:  sentence = ['how','do','you','do']
In[49]:  sentence.index('do')
In[50]:  1
In[51]:  sentence.index('who')
Out[52]: ValueError: 'who' is not in list
```

（9）**reverse 方法**：对列表元素倒序排列。

【例 3-20】

```
In[53]:  name = ['M','i','c','k','e','y']
In[54]:  name.reverse()
```

```
In[55]:  name
Out[55]: ['y', 'e', 'k', 'c', 'i', 'M']
```

3. 特殊的列表——元组

元组与列表一样，也是序列的一种，而区别就在于元组是不能修改的。创建元组方法很简单，只要将一些值用逗号分隔，就能自动创建。

【例 3-21】

```
In[56]:  1,2,3
Out[56]: (1, 2, 3)
```

同样还可以用圆括号括起定义元组。

【例 3-22】

```
In[57]:  (1,2,3)
Out[57]: (1, 2, 3)
```

当定义只包含一个值的元组时需要在它的后面加上逗号。

【例 3-23】

```
In[58]:  31
Out[58]: 31
In[59]:  31,
Out[59]: (31,)
In[60]:  (31,)
Out[60]: (31,)
```

元组并不太复杂，它的创建和访问与序列一致，除此之外可对元组执行的操作不多。

3.2.2　字符串

1. 字符串的概述

字符串在 Python 语言中很常见，它是一种最常用的数据类型。创建字符串很简单，只需要在引号('或")内赋值一个由字符数字等组成的变量即可。

【例 3-24】

```
In[1]:  name1 = 'Kitty'
In[2]   name1
Out[2]: 'Kitty'
In[3]   name2 = "Tom"
In[4]   name2
Out[4]  'Tom'
```

除此之外，字符串本身也是一种序列。因此，适用于序列的操作也同样适用于字符串，可以通过下标索引以及方括号截取字符串的方式访问字符串。对字符串的更新、修改和删除等操作，这里仅通过表 3-1 进行简单的说明。

表 3-1　字符串的基本操作

操作符	描　　述	实　　例
+	字符串连接	In[5]: name1+name2 Out[5]: 'KittyTom' *
*	重复输出字符串	In[6]: name2 * 2 Out[6]: 'TomTom'[]
[]	通过索引获取字符串中的字符	In[7]: name2[2] Out[7]: 'm'[:]
[:]	截取字符串中的一部分	In[8]: name1[2:4] Out[8]: 'tt'in
in	如果字符串中包含给定的字符返回 True	In[9]: 'K' in name1 Out[9]: Truenot in
not in	如果字符串中不包含给定的字符返回 True	In[10]: 'K' not in name2 Out[10]: True

2. 字符串用法

由于字符串很常用，因此字符串的使用方法也非常多，这里列一些较为常用的字符串使用方法来举例说明。

(1) **find 方法**：使用 find()函数在字符串中查找子串，如果找到，就返回子串的第一个字符的位置，未找到则返回－1。当出现返回值为 0 时，表明恰巧在最开始处找到了指定的子串。

【例 3-25】

```
In[11]:  sentence = 'Actions speak louder than words.'
In[12]   sentence.find('speak')
Out[12]: 8
In[13]   sentence.find('pick')
Out[13]  -1
In[14]   sentence.find('Actions')
Out[14]  0
```

同时，find 方法的第二个和第三个参数分别代表搜索的起点与终点，通过起点与终点，指明了搜索范围。需要注意的是，范围包含起点但不包含终点。

【例 3-26】

```
In[15]  sentence.find('s',3,10)
Out[15] 6
```

（2）**join 方法**：join 方法用于合并字符串，需要注意的是所有合并的序列元素都必须是字符串。

【例 3-27】

```
In[16]:  number_list = ['1','2','3','4','5']
In[17]   add = '+'
In[18]:  add.join(number_list)
Out[18]  '1+2+3+4+5'
```

（3）**split 方法**：split 是非常重要的字符串方法，它的作用与 join 方法相反，用于将字符串按照分隔符拆分为序列。如果没有指定分隔符，则在单个或多个连续的空白字符处进行拆分。

【例 3-28】

```
In[19]:  '1+2+3+4+5'.split('+')
Out[19]  ['1', '2', '3', '4', '5']
In[20]:  "I am a student from BUPT".split()
Out[20]  ['I', 'am', 'a', 'student', 'from', 'BUPT']
```

（4）**strip 方法**：使用 strip 方法将字符串开头和末尾的空白删除，并返回删除后的结果，需要注意的是 strip()函数删除并不会删除字符串中间的空格。同样，在 strip()函数中指定字符则可以删除在开头或末尾的对应字符，而中间的字符不会被删掉。

【例 3-29】

```
In[21]:  "      I am a student from BUPT       ".strip()
Out[21]  'I am a student from BUPT'
In[22]:  '*****!!!!!Something important!!!!!*****'.strip('*!')
Out[22]  'Something important'
```

strip()函数在处理数据中十分常见，当进行比较或切分操作之前，通常使用 strip()去掉不小心在尾部加上的空格，以免出现错误。

（5）**lower 方法**：用于将字符串中的大写字符全部替换为小写，当用户不想区分字符串大小写时很重要。例如用户名字、地址等信息往往大小写多样，而转换为字符串在存储与查找时，应先将它们全部处理为小写后，再进行操作。

【例 3-30】

```
In[23]:  'My favourite character is Mickey Mouse'.lower()
Out[23]  'my favourite character is mickey mouse'
```

3.2.3　字典

我们都知道序列使用索引来访问各个值，而 Python 中还提供一种数据结构，可以通

过名称来访问其各个值，这种数据结构就是映射（Mapping）。在映射中，较常用的就是字典，字典通过“键-值”对进行索引，这里的键可能是数、字符串或元组等。

字典的“键-值”对是它的特点，而“键-值”对也被称为项（Item）。每个键与其值之间都用冒号（：）进行分隔，各个项之间用逗号分隔，同时被包括在花括号内，类似于{key1：value1，key2：value2}的格式。为了实现字典中的快速查找，键必须是独一无二的，这样给定键就能直接找到该键对应的值，而值是可以重复的。

空字典中没有任何项，用两个花括号表示，类似于{}的格式。

下面详细介绍如何创建以及使用字典。

1. 创建字典

创建字典的最简单方式就是直接赋值，但需要注意的是保持键的唯一性，否则会导致后赋值的一个“键-值”对替换掉前面的。键的数据类型必须是不可变的，如字符串、数字或元组等，而值可以取任何数据类型。

【例 3-31】

```
In[1]:  number_dict = {'a':1,'b':2,'c':3,'c':4}
In[2]   number_dict
Out[2]  {'a': 1, 'b': 2, 'c': 4}
```

可以使用 dict 方法从其他的映射或“键-值”对来转换创建字典。

【例 3-32】

```
In[3]:  student = [('name','Mickey'),('age',24)]
In[4]   d = dict(student)
In[5]   d
Out[5]  {'age': 24, 'name': 'Mickey'}
```

2. 字典操作

字典也有索引、删除、修改对应项的值等操作，操作的基本思想与序列很像。下面通过举例说明。

【例 3-33】

```
In[6]:  student_dict={'Tom':24,'Mickey':23,'Marry':15,'Abel':18}
In[7]   student_dict
Out[7]  {'Abel': 18, 'Marry': 15, 'Mickey': 23, 'Tom': 24}
```

使用 len 方法可以计算字典中包含的“键-值”对数目。

【例 3-34】

```
In[8]  len(student_dict)
Out[8] 4
```

索引时使用键做下标来找到相应的值。

【例 3-35】

```
In[9]  student_dict['Marry']
Out[9] 15
```

对字典的值进行修改时，常使用赋值的方法。

【例 3-36】

```
In[10]  student_dict['Tom'] = 28
In[11]  student_dict
Out[11] {'Abel': 18, 'Marry': 15, 'Mickey': 23, 'Tom': 28}
```

使用 del 命令删除字典中对应的“键-值”对。

【例 3-37】

```
In[12]  del student_dict['Mickey']
In[13]  student_dict
Out[13] {'Abel': 18, 'Marry': 15, 'Tom': 28}
```

通过 in 命令来判断某一关键字是否在字典中。

【例 3-38】

```
In[14]  'Mickey' in student_dict
Out[14] False
```

从这些基本的操作来看，字典和列表的操作有很多相同之处，但也存在一些重要的不同之处。

首先，列表的索引只能对应于相应的位置，而字典中的键可以多种多样，并不是一定要设置为整数，只要保证字典的键是不可变的，则实数、字符串、元组等类型都可以。

其次，通过赋值列表无法增加新的项，即不能给列表中没有的元素赋值。但是即便是字典中原本没有的“键-值”，也可以通过赋值来创建新项。

【例 3-39】

```
In[15]  student_dict['Alan'] = 24
In[16]  student_dict
Out[16] {'Abel': 18, 'Alan': 24, 'Marry': 15, 'Tom': 28}
```

3. 字典方法

介绍了字典的操作后，接下来就介绍一些很有用的字典方法。

(1) **clear 方法**：使用该方法进行彻底地清除，删除所有的字典项，清除成功后返回值为 None。当然，除了使用 clear 方法外，还可以通过给字典赋空值来清空此字典。

【例 3-40】

```
In[17]  student_dict = {'Abel': 18, 'Alan': 24}
In[18]  student_dict.clear()
In[19]  student_dict
Out[19] {}
```

(2) **copy 方法**：使用该方法进行复制，返回一个新字典，其包含与原来的字典相同的“键-值”对。

【例 3-41】

```
In[20]  x = {'username': '201820091', 'grades': [90,87,10]}
In[21]  y = x.copy()
In[22]  y
Out[22] {'grades': [90, 87, 10], 'username': '201820091'}
```

(3) **get 方法**：get 方法是用来快速访问字典的项。访问 get 指定的键，与普通的字典查找结果一样，而使用 get 访问不存在的键是不会引发异常的，只是返回 None 值。

【例 3-42】

```
In[23]  d = {}
In[24]  print(d['name'])
Out[24] KeyError: 'name'
In[25]  print(d.get('name'))
Out[25] None
In[26]  d['name'] = 'Eric'
In[27]  d.get('name')
Out[27] 'Eric'
```

同时，get 方法还可以指定“默认”值，这样当未找到时，返回的将是指定的默认值而不是 None。

【例 3-43】

```
In[29]  d.get('name', 'N/A')
Out[29] 'N/A'
```

(4) **keys 方法**：函数 keys()返回一个字典的“键-值”视图，列出字典中的所有键。

【例 3-44】

```
In[30]  x = {'username': '201820091', 'grades': [90,87,10]}
In[31]  x.keys()
Out[31] dict_keys(['username', 'grades'])
```

(5) **pop 方法**：该方法首先找出指定“键-值”对，然后将它们从字典中删除。

【例 3-45】

```
In[32]  d = {'x': 1, 'y': 2}
In[33]  d.pop('x')
Out[33] 1
In[34]  d
Out[34] {'y': 2}
```

(6) **items 方法**：函数 items()将字典转换成列表，返回一个包含所有字典项的列表，其中每个元素都为(key，value)的形式。字典项在列表中的排列顺序不确定。返回值的类型为字典视图，可以对其执行成员资格检查。

【例 3-46】

```
In[35]  student = {'year': '2015', 'grades': [90,87,10]}
In[36]  student.items()
Out[36] dict_items([('year', '2015'), ('grades', [90, 87, 10])])
In[37]  len(student.items())
Out[37] 2
In[38]  ('year', '2015') in student.items()
Out[38] True
```

3.2.4　条件与循环语句

1. 条件与条件语句介绍及用法

目前为止的语句都是逐条执行的，而 Python 的条件语句通过某些条件是 True 或 False 来决定是否选择执行或是跳过某些特定的语句块。通常来说，False、None、各种类型(包括浮点数、复数等)的数值 0、空序列(如空字符串、空元组和空列表)以及空映射(如空字典)这些都被视为假，其他各种值都被视为真。

Python 中的条件语句用 if 和 else 来控制程序的执行，基本形式为

```
if 判断条件:
    执行语句段 1……
else:
    执行语句段 2……
```

当判断条件为真时，执行紧接着的语句段，这部分内容可以为多行，以缩进来区分。else 部分为可选语句，当有需要在条件不成立时执行的内容可以在此处写下。

if 语句的判断条件可以用>(大于)、<(小于)、==(等于)、>=(大于或等于)、<=(小于或等于)、!=(不等于)来表示。

Python 与 C 语言的条件语句有些不同，Python 不支持 switch 语句，所以多个条件判断，只能使用 elif 来实现。因此，当判断条件为多个值，可以使用以下形式。

```
if 判断条件 1:
```

```
    执行语句段 1…
elif 判断条件 2:
    执行语句段 2…
elif 判断条件 3:
    执行语句段 3…
else:
    执行语句段 4…
```

如果多个条件需要同时判断时，可以使用 or(或)，表示两个条件只要有一个成立时值即为真；使用 and(与)时，表示只有两个条件同时成立的情况下，值才为真。同时可使用圆括号来区分判断的先后顺序，圆括号中的判断优先执行，此外 and 和 or 的优先级低于>(大于)、<(小于)等符号，即大于和小于在没有括号的情况下会比 and 和 or 要优先判断。

除此之外，if 语句还可以实现嵌套，可将 if 语句放到其他 if 语句中，来实现判断后的再次判断。

下面通过一个使用条件语句的程序例子来说明。

【例 3-47】

程序 3-1　条件语句的应用实例

```
num = 9
if num < 0:
    print('负数!')
elif num > 0:

    # Python 3 中,下面式子括号里可合并为 0<=num <=100
    if (num >=0 and num <=100) or (num >=100 and num <=150):
        print(num)
else:
    print('大于 150!')
```

2. 循环语句介绍及用法

当了解了条件语句如何使程序在条件为真(或假)时执行特定的操作，接下来则进一步了解如何重复执行某些操作，这里通过循环语句来解决这个问题。

Python 的基本循环语句是 while 循环和 for 循环，在这些基础上加入一些跳出循环的语句就可以使得整个程序的流程更为多样，可能性更为丰富。

下面对这些一一进行介绍。

1) while 循环

while 循环的基本形式为

```
while 判断语句:
    执行语句…
```

执行语句可以是单个语句或语句段，判断语句可以是任何表达式，结果为真时就执行语句，当判断条件为假时，循环结束。

不仅如此，while 语句后还可以加 else 语句，表示在循环条件为假时执行另一段程序，形式为

```
while 判断语句:
    执行语句 1…
else:
    执行语句 2…
```

【例 3-48】

程序 3-2　while 循环的应用实例

```
numbers = [12,23,34,45,56,67,78,89]
even = []
odd = []
while len(numbers) >0:
    number = numbers.pop()
    if(number %2) == 0:
        even.append(number)
    else:
        odd.append(number)
else:
    print(even)
    print(odd)
```

输 出 结 果

```
[78, 56, 34, 12]
[89, 67, 45, 23]
```

2）for 循环

for 循环的基本思想是在遍历某一序列的每一个项目时执行一段语句，序列可以是一个列表或字符串，也可以是一个范围。

for 循环的基本形式为：

```
for 迭代变量 in 序列:
    执行语句…
```

当 for 循环遍历的是一个范围时，迭代变量相当于一个计数器，序列则通常是使用 len()函数获取的列表长度和 range()函数返回的元素范围等。

与 while 循环相类似，for 循环也可以同 else 结合表示跳出循环后执行的内容。

【例 3-49】

程序 3-3　for 循环的应用实例

```
books = ['math','chinese','computer science']
for index in range(len(books)):
    print("book about "+str(index)+":", books[index])
else:
    print("Bye")
```

输 出 结 果

```
book about 0: math
book about 1: chinese
book about 2: computer science
Bye
```

3）嵌套循环

Python 允许在一个循环体中嵌入另一个循环，例如同类型的，在 for 循环中嵌入 for 循环，在 while 循环中嵌入 while 循环。不仅如此，在 while 循环中也可以嵌入 for 循环，在 for 循环中也可以嵌入 while 循环等。

break 跳出循环：Python 的 break 语句用来终止循环语句，即使循环条件没有 False 条件或者序列还没被完全递归完，也会停止执行循环语句。它在 while 和 for 中都可以被使用，当使用嵌套循环时，break 语句将停止执行最深层的循环，并开始执行下一行代码。

【例 3-50】

程序 3-4　break 跳出循环的应用实例

```
for letter in 'Python':
   if letter == 'h':
      break
      print('这是 break 块')
   print('当前字母 :', letter)
print('break')
```

输 出 结 果

```
当前字母 : P
当前字母 : y
当前字母 : t
break
```

continue 跳出循环：相比于 break 跳出整个循环，continue 语句用来跳出本次循环。言下之意是，continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。它可以在 while 和 for 循环中使用。

【例 3-51】

程序 3-5　continue 跳出循环的应用实例

```
for letter in 'Python':          #第一个实例
   if letter == 'h':
      continue
      print('这是 continue 块')
   print('当前字母 :', letter)
print('continue')
```

输 出 结 果

```
当前字母 : P
当前字母 : y
当前字母 : t
当前字母 : o
当前字母 : n
continue
```

pass 跳出循环：Python 的 pass 是空语句，为了保持程序结构的完整性而设计的。pass 不做任何事情，一般用作占位语句。

【例 3-52】

程序 3-6　pass 跳出循环的应用实例

```
for letter in 'Python':
   if letter == 'h':
      pass
      print('这是 pass 块')
   print('当前字母 :', letter)
print('pass')
```

输 出 结 果

```
当前字母 : P
当前字母 : y
当前字母 : t
这是 pass 块
当前字母 : h
当前字母 : o
当前字母 : n
pass
```

3.2.5 函数

1. 定义函数

函数是预先设计好的、可重复使用的、能够实现单一或多种功能的综合代码段。使用了函数的 Python 代码，模块性较强，代码的重复利用率较高。目前为止我们可能或多或少地也使用过 Python 自带的函数，如 print()等。

当然，除了使用 Python 提供的函数以外，用户也可以自己定义，但是首先要知道的是，定义函数有如下几条规则。

首先，定义函数的标志关键字是 def，在它之后应该紧接着用户想要设计的函数名称。

其次，用户想要传入的参数用圆括号括起放在函数名后，并且在圆括号结束的位置之后加上冒号作为函数内容的开始。

同时，当开始定义函数内容时，要时刻保持缩进的使用是正确的。

最后，使用 return 语句表示函数结束时返回给调用者的值，不带 return 则相当于返回 None。

示例的结构如下：

```
def 函数名(参数列表):
    函数执行语句段…
    return [表达式]
```

2. 函数调用

当使用如上规则定义了一个函数，指定了函数名称、传递的参数以及代码块内容后，需要通过另外的语句调用函数，才能使这个函数在适当的时间与位置执行相应的功能。函数的调用可以嵌套在任何用户设定的位置里，如另外的函数中、命令行中等。下面举例说明函数的定义与调用过程。

【例 3-53】

程序 3-7 函数的定义与调用实例

```
def printstr(string):
    print(string)

printstr("函数的定义与调用!")
```

输 出 结 果

```
函数的定义与调用!
```

3. 参数传递

Python 的变量是没有特定类型的，定义的函数参数列表中的变量类型取决于参数传

递时所赋值的类型。同时根据函数传递的参数是否可以修改，将它们分成不可变类型与可变类型。

不可变类型参数传递本质上就是传值，常用的有数字、字符串等。当使用传值时，即使在函数内部对这些参数进行修改，它们的变化也不会影响函数外的值。因为当它们被传入函数后，相当于生成了另外一个复制的对象，使得函数内部语句在复制对象上进行操作。

可变类型参数传递本质上是传引用。传引用传入的是变量的地址，而当函数内部得到变量的地址后，再进行的操作相当于对原变量进行修改，修改后函数外部的变量也会受到影响。

【例 3-54】

程序 3-8　传不可变对象的应用实例

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

b = 3
ChangeInt(b)
print(b)
```

输 出 结 果

```
3
```

【例 3-55】

程序 3-9　传可变对象的应用实例

```
def ChangeList(mylist):
    "修改传入的列表"
    mylist.append([1,2,3,4])
    print("函数内取值: ", mylist)

mylist = [10,20,30]
print("更改前取值: ", mylist)
ChangeList(mylist)
print("更改后取值: ", mylist)
```

输 出 结 果

```
更改前取值:  [10, 20, 30]
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
更改后取值:  [10, 20, 30, [1, 2, 3, 4]]
```