第 3 章

Python 编程基础

本章主要为没有接触过或者正在学习 Python 编程的读者而设。不同于那些完整介绍 Python 编程的书籍和课程，作者并没有做铺陈的打算，而是尝试从本书的总体需求和特点出发，去粗取精，提炼“干货”，向读者介绍足以用来理解并且实践本书全部代码的 Python 基础编程知识。

3.1 Python 编程环境配置

在开始学习具体的 Python 编程知识之前，需要先解决下面两个问题：“Python 代码在哪里写？又在哪里运行？”

正所谓“工欲善其事，必先利其器”，目前在市场上，为 Python 代码提供编写和运行环境的应用程序非常多。如果从是否具备实时的交互性、是否能够支持远程的编写和运行服务、是否提供便捷开发的集成化功能和界面这三个维度来考量，那么 Python 的主流编程环境可以分为三种，分别是基于命令行/终端的交互式编程环境、基于 Web 的交互式编程环境，以及集成式开发环境。

接下来的各小节将分别就上述各个类别的开发环境进行举例说明。

3.1.1 基于命令行/终端的交互式编程环境

基于命令行/终端的交互式编程环境是我们最容易接触到的。如图 3.1 所示，Python 解释器本身就内置了一个交互式的编程环境。我们可以在 Windows 的命令提示符模式或者 macOS 和 Linux 的终端中，直接输入命令 python 调起 Python 解释器的默认编程环境。

这种交互式编程环境的优点在于，几乎输入的每一行 Python 代码都会立刻得到程序运行的反馈，便于人们了解每一行代码的执行情况。但是，这样做的缺点也非常明显：许

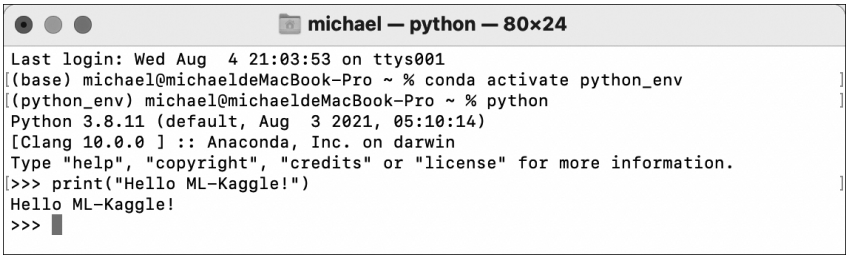

图 3.1 Python 内置的交互式编程环境

多大型程序的逻辑复杂，需要编写大量的代码，而不需要逐行查验程序输出。因此，这种交互式编程环境适合培养新手入门，但是熟练的程序员却不常使用这种编程环境。

与 Python 解释器编程环境类似的代表性应用有 IPython，如图 3.2 所示。比起图 3.1 的 Python 内置编程环境，IPython 算是一种增强型的终端交互式编程环境，对于一些 Python 编程的语法、关键字和内置函数，都会用不同颜色的字符进行突出显示。

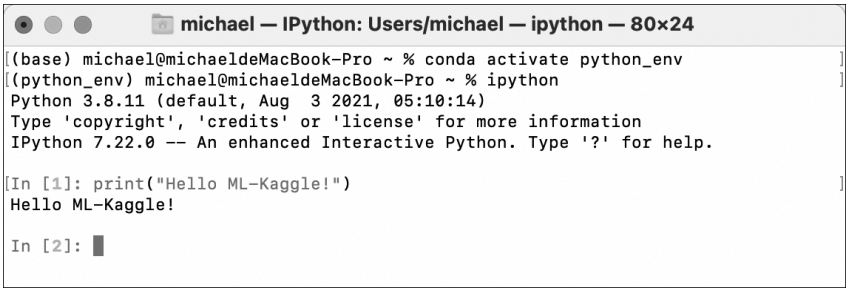

图 3.2 交互式编程环境 IPython

3.1.2 基于 Web 的交互式开发环境

除了基于终端的交互式开发环境，目前基于 Web 的交互式开发环境更加流行。Jupyter Notebook 就是这种 Python 开发环境的典型应用。如图 3.3 所示，这种类似于笔记一样的编程平台，也和 IPython 类似，能够立刻执行输入的程序，给出反馈。更进一步地，对于大段的 Python 程序，也可以将其整体输入 Jupyter Notebook 的文件中，并且整段运行。

因此，Jupyter Notebook 从一定程度上避免了 IPython 的缺点，能够既保留交互式编程环境的优点，同时也适用于大型程序的编写。但是，对于更加专业的程序设计人员，Jupyter Notebook 仍然无法提供更加高级的服务，如代码调试、内存中变量查看等其他

功能。

图 3.3　基于 Web 的交互式开发环境 Jupyter Notebook

3.1.3　集成式开发环境

集成式开发环境提供了更加专业的编程平台，除了能够为程序设计人员提供大型程序所需要的源代码编写空间以外，同时也集成了诸如代码调试、内存变量管理、单元测试等大量专业编程所需要的关键辅助功能。代表性的 Python 集成式开发环境包括 PyCharm 和 Spyder，分别如图 3.4 与图 3.5 所示。

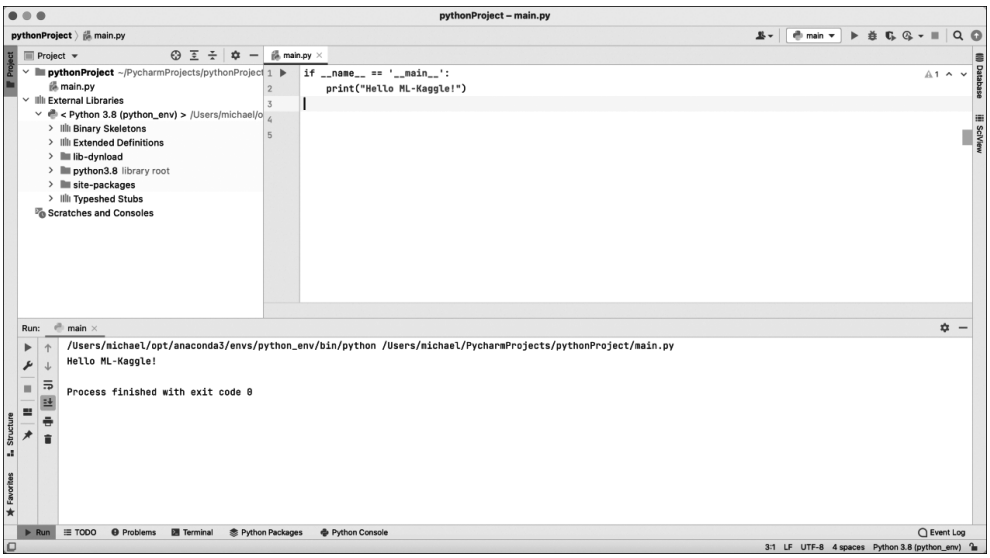

图 3.4　集成式开发环境 PyCharm

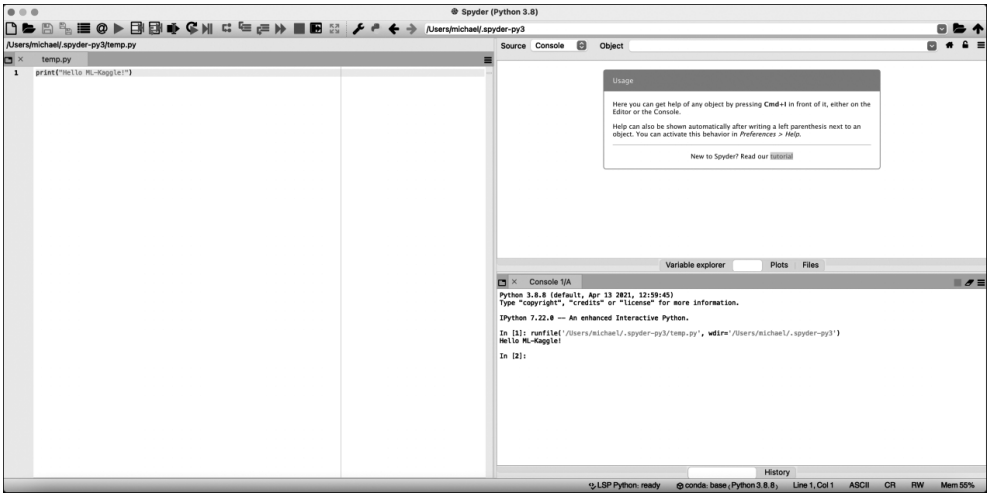

图 3.5　集成式开发环境 Spyder

3.2　Python 基本语法

“简单、易学、优雅”是本书对于 Python 编程基础的评价。Python 代码的可读性和撰写效率在大量的编程语言（Java、C++ 等）中是首屈一指的。

毫不夸张地说，即便是没有任何编程基础的读者，初看一些 Python 编程的基础代码，相信也能够勉强看懂其中的要旨。许多使用 Python 编写的程序只需要寥寥数行，就能够实现十分强大的功能。上述这些特点得益于 Python 基础语法特性。

3.2.1　赋值

对程序变量进行赋值声明是任何编程语言中都要进行的基本操作。如果在程序中贸然访问一个没有进行赋值声明的变量，如代码 3.1 所示，程序则会报错。

代码 3.1　Python 变量赋值的错误方式

```
In[*]:  is_ML_Geek

Out[*]: ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-2-97b21c2ef31d> in <module>
```

```
      1 #直接查验未定义变量 is_ML_Geek 的赋值,运行后程序报错
----->2 is_ML_Geek

NameError: name 'is_ML_Geek' is not defined
```

许多主流的高级编程语言,如 C++、Java 等,都需要在对变量进行赋值时声明变量的数据类型。但是,Python 不需要这样做。如代码 3.2 所示,Python 中对变量的赋值声明十分简单而且直接,只需要把具体的值通过“=”赋予所定义的变量名即可。

代码 3.2　Python 变量赋值的方式

```
In[*]:  is_ML_Geek =True

        is_ML_Geek
Out[*]: True
```

Python 变量名的定义有一套十分明确的规则:

- 变量名可以由字母、数字,或者下画线(_)组成,其中数字不能作为开头;
- 变量名不能是 Python 的关键字,但可以包含关键字;
- 变量名不能包含空格。

其中,Python 的关键字一般为许多内置的语句或者命令,如 if、for、in 等。这些关键字在 Jupyter Notebook 等专业编程环境中,一般都会用特殊的颜色突出显示,因此非常容易规避。另外,只要在变量声明时考虑其未来的可读性,尽可能贴近变量的实际意义进行命名,基本上都不太可能出现变量声明方面的错误。

3.2.2　注释

注释是代码可读性的另一层保障。与可执行的程序代码不同,注释不会被执行。如代码 3.3 所示,Python 编程语言的注释类型有两种:一种是单行注释,以“#”为行首,引导整行的内容作为注释,不会被 Python 解释器执行;另一种是多行注释,使用三个单引号或者三个双引号作为起始,直到对应的三个单引号或者双引号出现为止,中间的所有内容均被视为注释。

代码注释是一种优良的编程习惯,也是一种程序员之间的“礼仪”,于人于己都有好处,这个习惯非常建议读者保持。

代码 3.3 Python 注释样例

```
In[*]: #这是一个单行注释

       '''
       这是使用三个单引号的
       多行注释
       '''

       """
       这是使用三个双引号的
       多行注释
       """
```

3.2.3 缩进

代码缩进是 Python 语言的特色规则之一。许多其他的高级编程语言（如 C++、Java 等）不太关注代码的空白部分，代码的逻辑分割也大多使用其他标识符，如花括号、逗号等。这些用于分割逻辑的标识符占用了过多的编程空间。Python 则采用 Tab 缩进符，分割代码的逻辑层次，显著减少了源代码的长度。

如代码 3.4 所示，if 作为一个分支语句的关键字，提示后续的代码与其有从属的逻辑层次，因此需要进行缩进操作。其他使用缩进 Tab 符的常见场景还包括 for/while 循环语句、函数和类的声明等。

代码 3.4 Python 语句缩进的代码

```
In[*]: """
       如果变量 is_ML_Geek 的赋值为 True，那么程序会输出："推荐您购买《Python 机器学习实践：从零开始通往 Kaggle 竞赛之路》!"
       """

       if is_ML_Geek:
           print('推荐您购买《Python 机器学习实践：从零开始通往 Kaggle 竞赛之路》!')

Out[*]: 推荐您购买《Python 机器学习实践：从零开始通往 Kaggle 竞赛之路》!
```

3.3 Python 数据类型

Python 内置的常用数据类型共有 7 种，有简单的数值、布尔值、字符串，还有复杂一些的元组、列表、集合，以及字典。

另外，可以使用 type()函数来直接查验具体的数据类型。比起其他的高级编程语言，Python 内置数据类型的种类简化了许多，具体介绍如下。

- 数值(Number)。常用的数值类型包括整型数(Integer)、浮点数(Float)以及复数(Complex)。整型数和浮点数是我们平时最常使用的两类，如代码 3.5 所示，读者可以先简单地理解常用的整数，如 10、100、－100 等都是整型数；而一般用于计算的小数，如－0.1、10.01 等都可以使用 Python 的浮点数进行存储。复数在 Python 中不太常用，因此不过多介绍。

代码 3.5　Python 常用数值类型样例

```
In[*]:  type(10)          #整型数
Out[*]: int
In[*]:  type(0.34)        #浮点数
Out[*]: float
In[*]:  type(1+3j)        #复数
Out[*]: complex
```

- 布尔值(Boolean)。计算机的计算基础是二进制，因此任何一门编程语言都会有布尔类型，用来表示真/假。如代码 3.6 所示，在 Python 中，这两个值有固定的表示：True 代表真，False 代表假。请读者切记，Python 是大小写敏感的编程语言，因此只有按照 True 或者 False 这样严格区分大小写的输入才会被解释器理解为布尔值。

代码 3.6　Python 布尔值类型样例

```
In[*]:  type(True)        #布尔值:真
Out[*]: bool
In[*]:  type(False)       #布尔值:假
Out[*]: bool
```

- 字符串(String)。字符串是由一系列字符(Character)组成的数据类型，应用范围十分广泛，适用于文本数据的处理。如代码 3.7 所示，在 Python 中，字符串可以使用成对的英文单引号或者双引号进行表示，如'abc'或"123"。尽管 123 看似一个整型数，但是一旦被成对的单引号或者双引号限制起来，便成为了字符串类型的数据。

代码 3.7　Python 字符串数据类型样例

```
In[*]:  type('abc')    #用单引号声明的字符串
Out[*]: str
In[*]:  type("123")   #用双引号声明的字符串
Out[*]: str
```

上述 3 类都是 Python 基本的内置数据类型，它们是数据表达与存储的基础。下面即将介绍的 4 种数据类型则相对复杂，而且还需要上述 3 种基础数据类型的配合。

- 元组（Tuple）。元组是一系列 Python 数据类型按照顺序组成的序列，使用一组圆括号()表示。如代码 3.8 所示，(10，0.34，True，'abc')是一个包含 4 个元素的元组。读者会发现，元组中的数据类型不必统一，这是 Python 的一大特点。另外，x_tuple[0]的值为 10，x_tuple[1]的值为 0.34。也就是说，我们可以通过索引直接从元组中找到所需要的数据。特别需要提醒读者的是，大多数编程语言都默认索引的起始值为 0，而不是 1。

代码 3.8　Python 元组数据类型样例

```
In[*]:  x_tuple = (10, 0.34, True, 'abc')    #将一个元组赋值给变量 x_tuple
        print(x_tuple)
Out[*]: (10, 0.34, True, 'abc')
In[*]:  type(x_tuple)      #查验变量 x_tuple 的数据类型
Out[*]: tuple
```

- 列表（List）。列表和元组在功能上是类似的，只是表示方法略有不同。如代码 3.9 所示，列表使用一对方括号[]来组织数据，如：[10，0.34，True，'abc']。读者需要记住一点例外：Python 允许在使用者在访问列表的同时修改列表里的数据，而元组则不然。相关的样例代码会在 3.4 节中具体展示。

代码 3.9　Python 列表数据类型样例

```
In[*]:  x_list = [10, 0.34, True, 'abc']    #将一个列表赋值给变量 x_list
        print(x_list)
Out[*]: [10, 0.34, True, 'abc']
In[*]:  type(x_list)          #查验变量 x_list 的数据类型
Out[*]: list
```

- 集合(Set)。Python 使用一对花括号{}来初始化一个集合类型的数据。与列表和元组不同,集合中不允许出现相同的元素。如代码 3.10 所示,即便初始化一个集合时有重复的元素,但是,最终的集合只留下了不同的元素。

代码 3.10　Python 集合数据类型样例

```
In[*]:  x_set ={10, 0.34, 'a', 'a', 'b'}   #将一个集合赋值给变量 x_set
        print(x_set)
Out[*]: {0.34, 10, 'b', 'a'}
In[*]:  type(x_set)   #查验变量 x_set 的数据类型
Out[*]: set
```

- 字典(Dictionary)。字典是 Python 中非常实用而且功能强大的数据类型,特别在数据处理任务中,字典几乎成为了数据存储的主流形式。从字典自身的数据结构而言,它包括多组键(key)-值(value)对。如代码 3.11 所示,Python 的字典数据类型使用花括号{}来容纳这些键-值对,并且用英文冒号将键和值进行映射。特别需要读者注意的是,字典中的键是唯一的,但是没有数据类型的限制。而查找某个键所对应的值也和访问元组或者列表中元素的方式类似。例如,代码 3.11 中字典变量 x_dict 的键包括 1、2、'a'、'abc';x_dict['a']的值为 True。

代码 3.11　Python 字典数据类型样例

```
In[*]:  #将一个字典赋值给变量 x_dict
        x_dict ={1:10, 2:0.34, 'a':True, 'abc':'abc'}
        print(x_dict)
Out[*]: {1: 10, 2: 0.34, 'a': True, 'abc': 'abc'}
In[*]:  #查验变量 x_dict 的数据类型
        type(x_dict)
Out[*]: dict
```

3.4　Python 数据运算

3.3 节介绍了 Python 的数据类型,接下来,我们开始了解如何对这些数据进行运算。Python 中常用的数据运算类型有如下几种。

- 算术运算(Arithmetic Operators)。毫无疑问,算术运算是作为一门编程语言所必

须具备的基础运算功能。Python 中常用的算术运算符包括加法（+）、减法（-）、乘法（*）、除法（/）、取余（%），以及幂（**）。代码 3.12 展示了各种算术运算的样例。

代码 3.12　Python 算术运算样例

```
In[*]:  10 +20.0  #整型数与浮点数的加法,结果是浮点数
Out[*]: 30.0
In[*]:  30-60  #整型数与整型数的减法,结果是整型数
Out[*]: -30
In[*]:  4 * 8     #整型数与整型数的乘法,结果是整型数
Out[*]: 32
In[*]:  4 * 8.9   #整型数与浮点数的乘法,结果是浮点数
Out[*]: 35.6
In[*]:  5 / 4     #整型数除法,无论是否能够整除,结果都是浮点数
Out[*]: 1.25
In[*]:  6 / 3     #整型数除法,无论是否能够整除,结果都是浮点数
Out[*]: 2.0
In[*]:  5 // 4   #除法,结果取整
Out[*]: 1
In[*]:  4 / 5     #除法,结果取整
Out[*]: 0
In[*]:  5%4      #取余运算
Out[*]: 1
In[*]:  2 * * 3   #幂运算
Out[*]: 8
```

- 比较运算（Comparison Operators）。如果说算术运算的返回值一般是数值类型，那么比较运算则反馈布尔值类型的结果。代码 3.13 展示了常用于比较运算的运算符和样例。

代码 3.13　Python 比较运算样例

```
In[*]:  10 <20  #判断是否小于
Out[*]: True
In[*]:  10 >20  #判断是否大于
Out[*]: False
In[*]:  10==10.0  #判断是否等于
Out[*]: True
In[*]:  10!=20  #判断是否不等于
Out[*]: True
```

- 赋值运算(Assignment Operators)。算术运算和比较运算都有一个共同的特点，即所有的运算结果都只是作为一次性的输出。然而，在更多情况下，我们需要对数据运算的中间结果进行存储，以备后续使用。因此，我们需要将一些数据赋值给自定义的变量。赋值运算除了可以用于基本的 Python 变量初始化，也可以对已有的变量值进行修改。如代码 3.14 所示，绝大多数的 Python 数据类型都可以通过赋值运算进行修改，但是元组是一个例外。因此，元组和列表最大的不同在于：元组类型的内部元素不可以被修改；而列表类型却可以。

代码 3.14　Python 赋值运算样例

```
In[*]:  t =(1, 'abc', 0.4)

        t[0] =2  #元组类型的内部元素不可以被修改
Out[*]: ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-16-64c6100d331f>in <module>
              5 t =(1, 'abc', 0.4)
              6
        ---->7 t[0] =2 #元组类型的内部元素不可以被修改

        TypeError: 'tuple' object does not support item assignment
In[*]:  l =[1, 'abc', 0.4]  #将一个列表赋值给变量 l

        l[0] =2  #试图改变列表 l 的第一个元素，并且重新赋值为 2
```

```
#对更新后的列表的第一个元素进行递增 1 的操作,并重新赋值
l[0] +=1

l[0]      #观察输出,结果应为 3
```

```
Out[*]: 3
```

- 逻辑运算(Logical Operators)。逻辑运算比较简单,共有三种：与(and),或(or),非(not)。逻辑运算所涉及的数据类型为布尔值,返回值也是布尔值。代码 3.15 展示了部分常见逻辑运算的规则和结果。

代码 3.15　Python 逻辑运算样例

```
In[*]:  True and True
```

```
Out[*]: True
```

```
In[*]:  #and 作为逻辑运算时,只要有一方为 False,结果即为 False
        True and False
```

```
Out[*]: False
```

```
In[*]:  #or 作为逻辑运算符时,只要有一方为 True,结果即为 True
        True or False
```

```
Out[*]: True
```

```
In[*]:  #or 作为逻辑运算符时,两方均是 False,结果才是 False
        False or False
```

```
Out[*]: False
```

```
In[*]:  #非(not)运算符用来反转布尔值
        not True
```

```
Out[*]: False
```

- 成员运算(Membership Operators)。成员运算是针对 Python 中较为复杂的数据结构而设立的一种运算,主要面向元组、列表、集合以及字典。如代码 3.16 所示,我们可以借助运算符 in,询问某个元素是否在元组、列表或者集合中出现,或者检视某个键是否在字典中存在。这是 Python 中非常实用且功能强大的运算,适用于数据处理任务。

代码 3.16　Python 成员运算样例

```
In[*]:  l =[1, 'abc', 0.4]        #查询 0.4 是否在列表 l 中

        0.4 in l
Out[*]: True
In[*]:  t =(1, 'abc', 0.4)        #查询 1 是否在元组 t 中

        1 in t
Out[*]: True
In[*]:  s ={1, 'abc', 0.4}      #查询'abc'是否在集合 s 中

        'abc' in s
Out[*]: True
In[*]:  d ={1: '1', 'abc': 0.1, 0.4: 80}      #查询 0.4 是否在字典 d 的键中

        0.4 in d
Out[*]: True
In[*]:  80 in d       #尽管字典 d 的值中有 80,但是 in 只负责键的查询

Out[*]: False
```

3.5　Python 流程控制

本章前几节的示例代码都是按照从前到后的常规顺序依次执行的。按照先后顺序执行是最为常见的 Python 程序执行流程。然而,在某些情况下,我们需要选择执行或者重复执行某些代码片段。这就需要通过一些特殊的流程控制,使得 Python 解释器具备可以跳跃甚至回溯部分代码的能力。比较常见的流程控制方式包括分支语句(if-elif-else)和循环控制(for/while)。

3.5.1　分支语句

很多情况下,我们需要程序根据不同的情况选择性地执行某一部分代码,这就需要分支语句的参与。与分支语句紧密相连的数据类型和操作类型分别是布尔值与逻辑运算。

分支语句的关键字包括 if、elif 和 else。常见的分支语句包括单独使用 if、if-else 搭

配，以及 if 搭配多个 elif-else 的情况。

如代码 3.17 所示，if-else 的搭配最终只能选择 if 或者 else 两个分支之一的语句执行；并且每一个分支语句都需要进行缩进处理。具体执行哪个分支，取决于 if 后续的布尔值或者逻辑运算的结果是否为真。

代码 3.17　Python 分支语句 if-else 样例

```
In[*]:  b =True

        if b:
            print("变量 b 是真。")
        else:
            print("变量 b 是假。")
```

```
Out[*]: 变量 b 是真。
```

if-elif-else 的搭配如代码 3.18 所示。Python 解释器会首先询问 if 对应的布尔值或者逻辑运算为布尔值的表达式。如果为 True，那么不会执行 elif 以及 else 对应的分支语句；如果为 False，则会依次试探 elif 对应的布尔值或者逻辑运算结果为布尔值的表达式。以此类推，一旦其中任何一个为真，便会执行对应的多行分支语句，然后跳出整体分支语句。如果 if 和 elif 中没有任何一个为真，则执行 else 对应的语句。

代码 3.18　Python 分支语句 if-elif-else 样例

```
In[*]:  a =False
        b =False
        c =True

        if a:
            print("变量 a 是真。")
        elif b:
            print("变量 a 是假，并且变量 b 是真。")
        elif c:
            """
            如果执行到此处，说明 a 一定是 False，同时 b 一定是 False，c 必须是真。
            这是因为，分支语句依次执行真假判别。一旦某一分支判别为真，则该条语句被执行，
        并且整体分支语句就会结束，根本没有机会执行后续其他的 elif 和 else。
            """
            print("变量 a 是假，同时变量 b 也是假，并且变量 c 是真，这条语句才会被执行。")
        else:
            print("所有变量都是假!")
```

```
Out[*]: 变量 a 是假,同时变量 b 也是假,并且变量 c 是真,这条语句才会被执行。
```

多个 if 引导的分支语句之间互不干涉,各自分别执行,如代码 3.19 所示。

代码 3.19　Python 多个分支语句样例

```
In[*]:  """
        换一种情况,两个 if 引导不同的两个分支语句,都会分别执行,互相不存在依赖关系。
        """

        a =False
        b =False
        c =True

        if not a:
            print("变量 a 是假。")

        if b:
            print("变量 b 是真。")
        elif c:
            print("变量 b 是假,变量 c 是真,这条语句才会被执行。")
        else:
            print("变量 b 和 c 都是假!")
Out[*]: 变量 a 是假。
        变量 b 是假,变量 c 是真,这条语句才会被执行。
```

3.5.2　循环控制

还有一些情况,需要我们循环执行某些代码。这是计算机程序为人类提供的极大便利,可以把某些具有规律的重复性代码单独封装。

循环控制可以根据需要采用 for 或 while 引导的两种不同的语法。如代码 3.20 所示,以关键字 for 引导的循环控制,需要搭配之前介绍的成员运算符 in 形成固定的语法结构,借助遍历来完成对循环语句的控制。

代码 3.20　Python 循环控制(for)样例

```
In[*]:  d ={1: '1', 'abc': 0.1, 0.4: 80}

        for k in d:
```

```
            print(k, d[k])
```

```
Out[*]: 1 1
        abc 0.1
        0.4 80
```

而以关键字 while 引导的循环控制，如代码 3.21 所示，则需要在每一轮判定其后续的布尔值或者逻辑表达式是否为真，从而决定是否要继续执行其循环语句。

代码 3.21　Python 循环控制（while）样例

```
In[*]: l = [1,2,3,4,5]
       i = 0
       while i < len(l):
           print(l[i])
           i += 1
```

```
Out[*]: 1
        2
        3
        4
        5
```

3.6　Python 函数设计

在面对大型项目的时候，编程人员不可避免地需要编写越来越多的代码。在实际编码过程中，人们往往会发现有很多功能相同的代码片段被反复地重用和执行。因此，人们开始考虑是否可以将这些功能明确且被经常使用的代码片段从程序中抽离出来单独封装。于是，函数（Function）的概念出现在了编程语言里。有了函数的帮助，我们便可以更好地组织和规划更加大型的项目。

在函数的设计方面，我们可以向函数提供必要的参数作为输入，同时也可以从函数获取所需要的返回值。Python 采用 def 这个关键字来定义一个函数。代码 3.22 用 def 定义了一个函数，名称为 foo，并且 foo()函数的参数为 x，返回值（return）为 x * * 2。

代码 3.22　Python 函数定义和执行样例

```
In[*]: def foo(x):
           """
           这是一个计算二次方的函数
           """
```

```
        return x * * 2


    if __name__ =='__main__':
        print(foo(8.0))
```

```
Out[*]: 64.0
```

3.7　Python 面向对象编程

Python 从设计之初就已经是一门面向对象的编程语言。与 C++、Java 等高级编程语言一样，使用 Python 创建一个类或者实例化一个对象，是很容易的操作。

我们首先要了解一些面向对象语言所包含的基本概念。

- 类(Class)。类是用来描述具有相同属性和方法的对象的集合。类是我们在日常生活中用来描述事物的抽象概念，例如猫、狗、学生、雇员等，都可以用来概括描述一类事物。为了更进一步描述类，我们通常需要在类中定义必要的数据成员和方法(函数)。
- 对象(Object)/实例(Instance)。对象/实例是通过类具象化定义的数据结构。例如，我们定义“范森”是一位“学生”，所以，“学生”类的一个对象或者具体实例是“范森”。而创建一个类的实例，即创建类的具体对象的过程称为实例化。沿用之前的例子，创建“范森”是对“学生”概念的实例化过程。
- 数据成员。数据成员用来定义类的相关数据，包括类变量、实例变量(属性)等。例如，对于“学生”这个类，通常我们需要记录学生的姓名、年龄、性别等属性(Attribute)，同时，这些属性会根据具体对象/实例的不同而存储不同的数值。
- 方法(Method)。方法是类里面所定义的函数。例如，“学生”这个类可以有入学、分班、上课、考试等方法，这些方法可以对每个对象自身的属性，或者结合其外部信息，给出必要的反馈。

接下来，我们通过一个有关学生信息处理的样例代码来具体说明上述有关 Python 面向对象编程的基本概念。代码 3.23 中定义了一个名为 Student 的类。

代码 3.23　Python 面向对象编程样例

```
In[*]:  class Student:
            count=0
            __tax_discount_rate=0.1
```

```
    def __init__(self, name, age, sex, fee, is_married):
        self.name=name
        self.age=age
        self.sex=sex

        self.__fee=fee
        self.__is_married=is_married

        Student.count +=1


    def display_student(self):
        print("姓名: %s, 年龄: %d, 性别: %s" %(self.name, self.age, self
.sex))


    def display_fee(self):
        print("%s 的学费为%f" %(self.name, self.__fee))


    def __get_is_single(self):
        if self.__is_married:
            return False
        else:
            return True


    def get_tax_return(self):
        if self.__get_is_single():
            return self.__fee * Student.__tax_discount_rate
        else:
            return 0.0


    def display_tax_return(self):
        print("%s 的退税额度为:%f" %(self.name, self.get_tax_return()))
```

```
if __name__ =='__main__':
    student_1 =Student("范淼", 29, "男", 5000.0, False)
    student_2 =Student("刘晓龙", 34, "男", 5500.0, False)

    student_1.display_student()
    student_2.display_student()

    print("目前学生总数为%d 人" %Student.count)

    student_3 =Student("孙华枭", 28, "男", 5600, True)
    student_4 =Student("陈蓓", 27, "女", 4500, True)

    print("目前学生总数为%d 人" %Student.count)

    student_1.display_tax_return()
    student_3.display_tax_return()
```

```
Out[*]: 姓名: 范淼, 年龄: 29, 性别: 男
姓名: 刘晓龙, 年龄: 34, 性别: 男
目前学生总数为 2 人
目前学生总数为 4 人
范淼的退税额度为:500.000000
孙华枭的退税额度为:0.000000
```

Student 类中包含了公开的属性 name、age、sex，以及私有的属性__fee 和__is_married。Student 类中也定义了公开的方法 display_student()、display_fee()、get_tax_return()和 display_tax_return()，以及私有的方法__get_is_single()。

从上述属性和方法的命名规则，我们可以看出，如果想要私有化属性或者方法，需要在属性和方法的前面增加双下画线“__”。

此外，我们还注意到 Student 类中有两个特别的地方：一个是__init__()方法，这是初始化对象或者实例时需要默认调用的方法，所有的公有和私有属性都在这个方法中初始化；另一个是在__init__()方法外侧的 count 公有变量和__tax_discount_rate 私有变量，这两个变量的值会在 Student 类的所有实例(对象)之间共享。

最后，需要强调的是，每一个类的方法在声明时，至少要引入 self 参数代表这个类本身，然后再声明其他的函数参数。

3.8 Python 编程库（包）导入

当各位读者已经学会如何通过封装和调用自己编写的函数来完成较为复杂的任务时，心中一定充满了成就感。特别是如果对于自己所编写的某些函数的功能和效率都信心十足，读者一定非常希望可以和其他人分享，并且也期望别人的程序重用这些函数模块。这种分享的动机激励着 Python 开发者互通有无，使得第三方编程库（Library）或者包（Package）的概念应运而生。有一些编程库默认配置在 Python 的解释器环境中，这些是我们经常要用到的；也有一些是由其他编程爱好者所开发的，发布在其他平台上。我们可以借助 Anaconda Navigator（如图 3.6 所示）或者 conda 命令（如图 3.7 所示，命令为 conda install matplotlib）安装这些编程库。

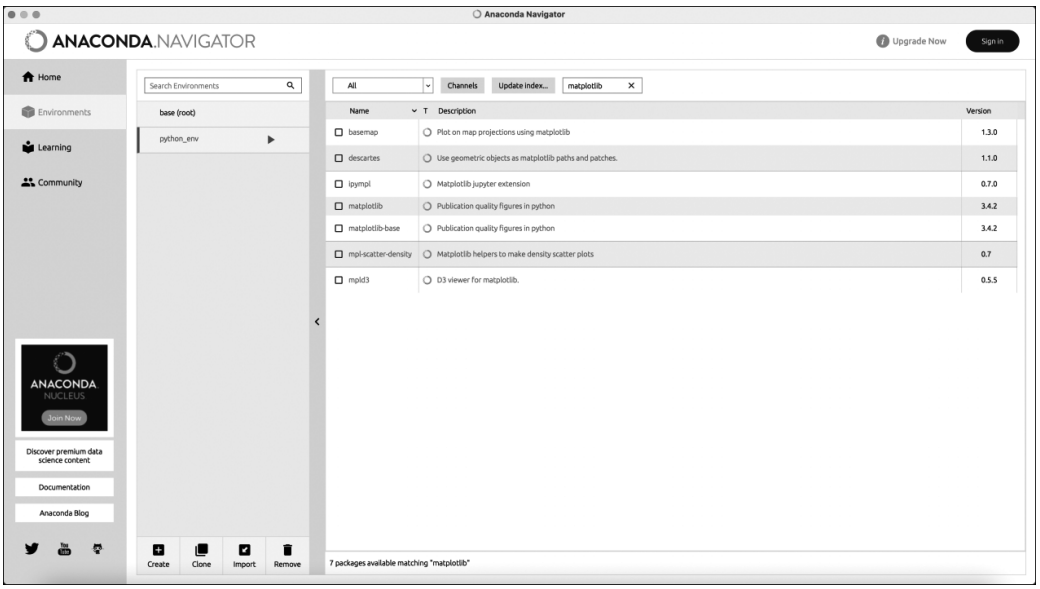

图 3.6 使用 Anaconda Navigator 在虚拟环境 python_env 中安装 matplotlib 编程库

事实上，越是复杂的大型项目越不可能从零开始编程，更不可能要求一位程序员自行编写一个大型项目所有功能的代码。实际使用中，哪怕是执行一些相对简单的数学运算，我们甚至都能在 Python 语言的内置编程库（包）中找到可以导入（import）的编程库。代码 3.24 列出了几种导入 Python 编程库（包）的方法。