第 1 章

Python基础知识

为了使本书自成一体，形成一个完整的学习环境，所以本章为没有 Python 基础的读者介绍学习本书后续内容所需掌握的 Python 基础知识和技术。为了使本章内容更有针对性，且避免内容冗余，本章只介绍与本书后续内容相关的 Python 编程知识，已具备 Python 编程经验的读者可以直接从第 2 章开始阅读。

1.1 在 Python 中安装 pip 程序

pip 是一个用于在 Python 中安装第三方软件包及其依赖项的管理程序，所有在 Python 中使用的第三方软件包，都需要在系统命令行窗口（例如 Windows 中的命令提示符）中使用 pip 程序进行安装。

在操作系统中安装 Python 解释器时，有个用于控制是否安装 pip 程序的选项，保持默认设置会自动安装 pip 程序。如果在安装 Python 解释器时没有安装 pip 程序，则可以在系统命令行窗口中输入以下命令来安装该程序，该命令中的 ensurepip 是 Python 标准库中的一个模块，专门用于单独安装 pip 程序。

```
python -m ensurepip
```

提示：Python 中的“包”是一个包含一个或多个模块的代码集合，使用这些模块可以实现一个复杂功能涉及的各部分子功能。模块是扩展名为 .py 的文件，模块中的代码可以被导入到任何 Python 程序中，便于代码的重复使用。可能在很多场合看到过“库”这个术语，库的功能与 Python 中的包类似，也是一个包含代码的集合，便于程序员重复使用这些代码，提高开发效率。库的一个示例是 Python 自带的标准库，其中包含大量的模块，用于实现各种功能。可以简单地将 Python 中的“包”等同于“库”，本书将混合使用这两个术语。

1.2 在 Python 中安装和导入软件包

在 Python 中可以将软件包安装到两种环境下；一种是全局环境，这是指在操作系统中安装的 Python 解释器的目录中安装软件包；另一种是虚拟环境，这是指将软件包安装到预先创建的 Python 虚拟环境中，这种环境与在操作系统中安装的 Python 解释器是隔离

开的。

无论在哪种环境中安装软件包，都需要使用 pip 程序从 PyPI（Python Package Index）网站自动下载和安装软件包。PyPI 是一个开源的 Python 资源库，来自全世界的 Python 程序员将他们创建的软件包发布到 PyPI 网站，供所有 Python 程序员免费使用。安装好软件包后，需要将其导入到 Python 中，然后才能使用软件包中的功能。

1.2.1 在全局环境中安装软件包

无论在全局环境中安装哪种软件包，都需要在系统命令行窗口输入以下命令（命令中的英文字母不区分大小写）。

```
pip install 软件包的名称
```

下面的命令用于安装 Matplotlib 软件包（也可称为 Matplotlib 库），在命令行中输入该软件包的名称时，所有英文字母必须小写。

```
pip install matplotlib
```

如果在操作系统中安装了多个 Python 版本，则当前运行的 pip 程序可能会将软件包安装到一个与该 pip 程序不匹配的 Python 版本中，这意味着没有将软件包安装到所希望的目标 Python 版本，导致在目标 Python 版本中无法使用该软件包。

避免上述问题的方法是在系统命令行窗口中使用以下命令安装软件包，通过 -m 参数调用与当前 Python 解释器关联的 pip 模块，确保使用与当前 Python 解释器关联的 pip 程序安装软件包。

```
python -m pip install 软件包的名称
```

如果只想为操作系统中的当前用户而非所有用户安装软件包，则可以在命令中添加 --user 参数。

```
python -m pip install --user
```

PyPI 网站中软件包的版本会不定期更新，如需将当前已安装的软件包更新到最新版本，可以在命令中添加 --upgrade 参数。

```
python -m pip install --upgrade 软件包的名称
```

1.2.2 一次性安装多个软件包

如需安装多个软件包，可以使用前面介绍的方法，逐个安装每一个软件包，即执行多次 pip 命令进行安装。另一种方法是将要安装的所有软件包的名称保存到 requirements.txt 文件中，每行一个名称，所有名称纵向排列。然后在系统命令行窗口中输入以下命令，将

自动安装 requirements.txt 文件中列出的所有软件包。如果 requirements.txt 文件不在当前工作目录中，则需要为其指定完整路径。

```
python -m pip install -r requirements.txt
```

提示：如果想让软件包的安装变得更简单，可以从 Anaconda 官方网站下载 Anaconda 安装程序，然后在操作系统中安装 Anaconda。Anaconda 内置了大量的在 Python 中使用的数据处理和分析、数据可视化和科学计算等方面的软件包，只要安装好 Anaconda，就无须额外安装这些软件包了。

1.2.3　创建和删除虚拟环境

如果经常使用大量不同类型的软件包，由于某些软件包之间不兼容，可能会出现 Python 解释器无法正常工作或其他一些无法预料的问题。只有从操作系统中卸载 Python 解释器，然后再重新将其安装到操作系统中，并进行诸如系统环境变量等的一系列配置，才能使 Python 解释器正常工作。

为了避免上述问题，可以将不同类型和用途的软件包分别安装到相互隔离的多个虚拟环境中。当某个软件包导致问题时，只需删除该软件包所在的虚拟环境即可，而不会影响其他虚拟环境，以及安装在操作系统中的 Python 解释器。

Python 标准库中的 venv 模块用于创建虚拟环境，在系统命令行窗口中输入以下命令，将在指定的路径中创建一个虚拟环境，该命令中的英文字母大小写均可。本例创建的虚拟环境的名称为 PyDataViz，路径中的文件夹不存在时会自动创建它们。

```
python -m venv E:\测试数据\Python\PyDataViz
```

虚拟环境在计算机中以文件夹的形式存在，路径中最后一个部分的名称就是用作虚拟环境的文件夹的名称，该名称也是虚拟环境的名称，如图 1-1 所示。

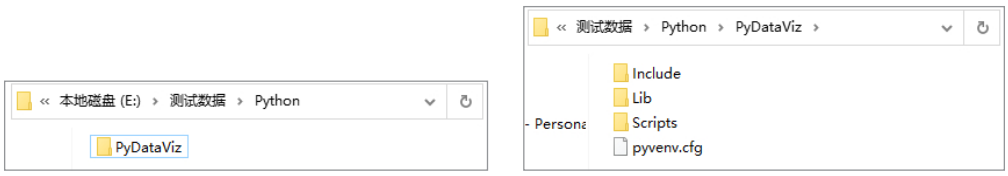

图 1-1　虚拟环境在计算机中的文件夹

如需删除虚拟环境，可以在文件资源管理器中删除虚拟环境所在的文件夹。

提示：还有很多用于创建虚拟环境的第三方工具，有兴趣的读者可以在 Internet 中搜索。

1.2.4　激活和退出虚拟环境

如需将软件包安装到 Python 虚拟环境中，需要先激活指定的虚拟环境。在系统命令行窗口中输入以下命令，将激活前面创建的名为 PyDataViz 的虚拟环境。activate.bat

命令位于虚拟环境所在的文件夹的 Scripts 子文件夹中。输入该命令时可以省略 .bat 扩展名。

```
E:\ 测试数据 \Python\PyDataViz\Scripts\activate.bat
```

激活虚拟环境后，将在提示符的开头显示虚拟环境的名称，如图 1-2 所示，以后输入的代码都将运行在该虚拟环境中。

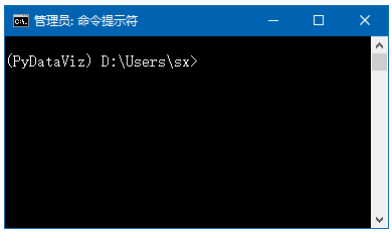

图 1-2　激活后的虚拟环境

如果只想输入 activate.bat 命令，而省略该命令前面的路径部分，则可以将该命令的路径添加到操作系统的 PATH 环境变量中。或者在系统命令行窗口中将提示符切换到 Scripts 文件夹，然后在提示符右侧输入 activate.bat 命令。

如需退出虚拟环境，可以在激活虚拟环境后，在系统命令行窗口中输入以下命令：

```
deactivate
```

1.2.5　在虚拟环境中安装软件包

激活虚拟环境后，在其中安装软件包的方法与在全局环境中安装完全相同，只需输入以下命令，即可将软件包安装到当前虚拟环境中。

```
python -m install 软件包的名称
```

提示：使用 requirements.txt 文件安装多个软件包的方法也适用于虚拟环境。

1.2.6　在虚拟环境中使用 IDLE

如需在虚拟环境中使用 IDLE，可以在系统命令行窗口中输入以下命令。打开 IDLE 窗口后，在其中输入的 Python 代码都运行在虚拟环境中。

```
python -m idlelib.idle
```

1.2.7　导入整个软件包

启动 Python 解释器，然后使用 Python 中的 import 语句，将指定的软件包导入到

Python 中。下面的代码在 Python 中导入 Matplotlib 软件包。

```
import matplotlib
```

导入后，可以在代码中使用 matplotlib 引用该软件包中的模块、类和函数，以便使用它们进行编程。

如果软件包的名称较长，为了简化输入，可以在导入软件包时为其设置别名，以后可以使用别名代替其原名。下面的代码导入 Matplotlib 软件包，并将其别名设置为 mpl，以后在代码中使用 mpl 代替 matplotlib。

```
import matplotlib as mpl
```

可以一次性导入多个软件包，各个软件包之间以逗号分隔。下面的代码导入 NumPy 和 Pandas 两个软件包。

```
import numpy, pandas
```

1.2.8　导入软件包中的特定模块

如果只使用软件包中的某个模块所提供的功能，则可以在 Python 中只导入该模块，从而避免代码变得混乱。下面的代码是在 Python 中导入 Matplotlib 软件包的 pyplot 模块，使用英文句点连接软件包的名称及其中的模块名称。

```
import matplotlib.pyplot
```

同样，可以为导入的模块设置别名。

```
import matplotlib.pyplot as plt
```

1.3　在交互模式和脚本模式中编写代码

可以在交互模式或脚本模式中编写 Python 代码。

1. 交互模式

使用系统命令行窗口或 Python 自带的 IDLE，可以在交互模式中编写 Python 代码。在交互模式中每输入一行代码并按 Enter 键，会自动运行代码并显示运行结果，如图 1-3 所示。

如果输入的是复合语句（显示在多行但被认为是同一组语句），则需要在一次性输入完所有这些语句后再执行它们。输入多行语句时，首行语句的开头显示主提示符（>>>），其他行语句的开头显示次要提示符（…），如图 1-4 所示。

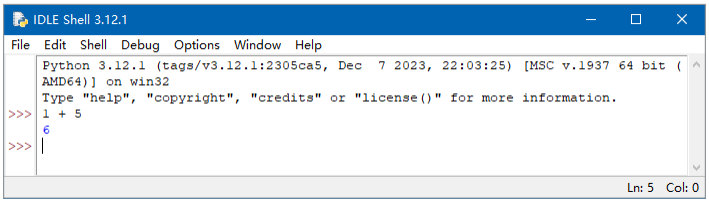

图 1-3　交互模式

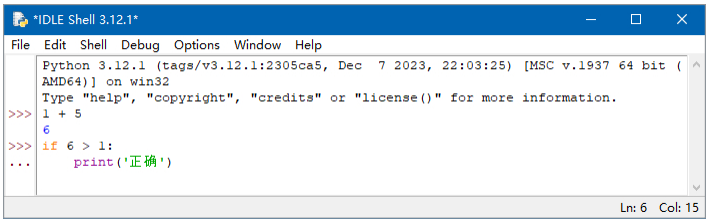

图 1-4　复合语句

2. 脚本模式

使用 Windows 记事本、IDLE 的文本编辑窗口或其他文本编辑工具，可以在脚本模式中编写 Python 代码，如图 1-5 所示。在脚本模式中可以编写任意行代码，编写过程中可以随时修改已编写好的代码，然后自顶向下运行每一行代码。与在交互模式中自动运行代码不同，在脚本模式中需要手动运行代码，例如，在 IDLE 的文本编辑窗口中可以按 F5 键运行代码。

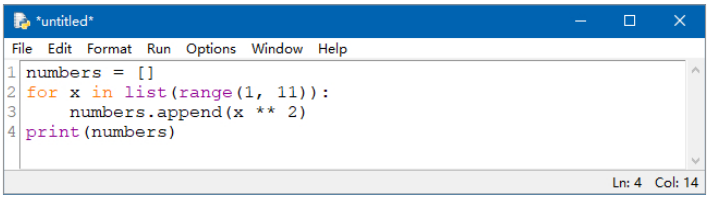

图 1-5　脚本模式

在脚本模式中编写的代码可以保存在扩展名为 .py 的文件中，以后可以继续编写或修改文件中的代码，还可以将 .py 文件以模块的形式导入到其他项目中。

1.4　在屏幕上打印数据

使用 Python 内置的 print 函数，可以将程序的运行结果或指定的数据打印到屏幕上。在脚本模式中编写代码时必须使用 print 函数，才能让数据显示在屏幕中，而在交互模式中即使不使用 print 函数，程序的运行结果也会自动显示出来。本节将介绍 pirnt 函数的常

见用法，以及如何对字符转义和抑制转义，还将介绍在程序中使用变量引用数据的方法。如无特别说明，本节中的示例代码都是在脚本模式中编写的。

1.4.1　打印数据的基本方法

将要打印的数据作为参数传递给 print 函数，即可使用 print 函数将该数据打印到屏幕上。下面的代码将“你好”两个字打印到屏幕上，即在屏幕上显示“你好”。代码中的“你好”是一个字符串，Python 代码中的任何字符串都必须放在一对单引号或双引号中。

```
print('你好')
```

可以一次性打印多个数据，将这些数据传递给 print 函数时，各个数据之间使用逗号分隔，表示向 print 函数传递多个参数。下面的代码在屏幕上打印 3 个数字，各个数字之间默认以空格分隔。

```
print(1, 3, 5)
```

代码的运行结果如下：

```
1 3 5
```

1.4.2　自定义数据之间的分隔符

打印多个数据时，可能想要以指定的符号分隔数据，此时可以为 print 函数指定 sep 参数，将该参数的值设置为所需的分隔符。下面的代码使用“-”符号分隔 3 个数字，sep 参数必须放在传递给 print 函数的所有数字之后。

```
print(1, 3, 5, sep='-')
```

代码的运行结果如下：

```
1-3-5
```

1.4.3　自定义数据末尾的终止符

下面的代码使用两个 print 函数将两组数字分别打印到两行中。

```
print(1, 3, 5, sep='-')
print(7, 9, 11, sep='-')
```

代码的运行结果如下：

```
1-3-5
7-9-11
```

第二个 print 函数打印的数字会自动显示到第二行，这是因为 print 函数有一个 end 参数，该参数用于指定打印的最后一个数据之后的字符。如果不指定该参数的值，则其值默认为换行符，所以上面示例中的第二个 print 函数打印的数据会被转到第二行。

如果显式指定 end 参数的值，则会在打印的最后一个数据之后添加特定的字符。下面的代码使用“-”符号将两个 print 函数打印的数字首尾相连。

```
print(1, 3, 5, sep='-', end='-')
print(7, 9, 11, sep='-')
```

代码的运行结果如下：

```
1-3-5-7-9-11
```

1.4.4 转义字符和抑制转义

在 Python 中，当一个反斜线和某个特定字母组合在一起时会产生特殊的含义。例如，“\n”表示换行符，“\t”表示制表符。这种现象经常出现在文件路径中，由于路径中的各个部分都以反斜线分隔，所以当路径中的某些部分以英文字母开头时，就有可能对首字母转义。

输入下面的代码不会得到正确的路径，因为第二个反斜线会将右侧的字母 n 转义，将其转换为换行符。

```
print('E:\测试数据\newPython')
```

运行上面的代码将显示以下结果，整个路径被分成上下两行，而且会丢失路径中的字母 n。

```
E:\测试数据
ewPython
```

让路径恢复正常显示的一种方法是，在整个路径的开头添加字母 R 或 r，这样可以阻止反斜线转义特定的字母，使路径中的每个字符保持原有含义。

```
print(r'E:\测试数据\newPython')
```

当反斜线位于路径的末尾时，即使添加字母 R 或 r，也无法得到正确的路径。此时可以将路径中的每个反斜线替换为两个反斜线，形式如下：

```
print('E:\\测试数据\\newPython\\')
```

1.4.5 使用变量引用数据

变量是一个由用户指定的名称，使用这个名称可以存储任何数据，可以是输入的字面

值，也可以是由数字、字符串、其他 Python 内置对象和运算符组成的表达式。下面的代码将数字 100 存储到名为 number 的变量中，然后在屏幕中打印该变量的值，将显示 100。

```
number = 100
print(number)
```

与很多编程语言不同，在 Python 中为一个变量赋值之前不需要先声明该变量。但是在后续代码中处理一个变量之前，必须先为该变量赋值，否则会导致程序出错。将值保存到变量中的操作称为“为变量赋值”，使用等号连接变量和值，变量在等号的左侧，值在等号的右侧。

提示：实际上，在 Python 中为变量赋值时，值本身并未真正保存到变量中，变量只是指向值在内存地址中的引用。

变量的名称应该能够反映出变量的含义或用途，这样可以增加代码的可读性。在 Python 中为变量命名时需要注意以下几点：

- 不能以数字开头。
- 只能使用大写或小写的英文字母、数字和下画线。
- 英文字母区分大小写。
- 类的首字母大写，函数和变量的首字母小写。这条规则只是大多数 Python 程序员约定俗成的，而非强制要求。

提示：编写 Python 代码时可能会遇到开头和结尾都带有下画线的变量，它们是 Python 内部定义的变量。

1.5　函数式编程

Python 是一种高度灵活的编程语言，无须声明即可直接使用变量就是其灵活性的体现，更大的灵活性体现在 Python 同时支持函数式编程和面向对象编程。本书后续内容介绍数据可视化编程时会使用这两种编程方式，所以本节和 1.6 节将分别介绍函数式编程和面向对象编程的基本方法。

1.5.1　使用 Python 内置函数

Python 内置了很多函数，可以完成一些常见任务。运行以下代码将显示一个列表，其中以小写字母开头的是 Python 内置函数。代码中的 builtins 是一个 Python 内置模块。

```
import builtins
print(dir(builtins))
```

在上面的第二行代码中使用了两个 Python 内置函数 --print 和 dir。print 函数在前面介

绍过，用于将指定的数据打印到屏幕上。dir 函数用于显示指定对象的所有属性。

在代码中使用一个函数称为调用函数。调用一个函数时，需要输入该函数的名称，然后在其右侧的一对小括号中输入函数的参数。参数是函数要处理的数据，上面示例中的 builtins 就是 dir 函数的参数。

调用一个函数通常会返回一个值，值的类型可以是数字、字符串或其他 Python 对象。例如，dir 函数返回一个 list（列表）对象，这个列表包含 buintins 模块的所有属性。

1.5.2 按照位置或关键字传递参数

调用函数时可以按照位置或关键字向函数传递参数。此处以 Python 内置的 pow 函数为例，介绍传递参数的两种方式。pow 函数用于计算指定数字的 n 次幂，语法格式如下：

```
pow(base, exp, mod=None)
```

pow 函数有 3 个参数，第一个参数 base 表示要计算幂的底数，第二个参数 exp 表示要计算幂的指数。这两个参数都是必选的，在调用 pow 函数时必须指定它们的值。第三个参数 mod 是可选的，在调用 pow 函数时可以不指定该参数的值以忽略它。如果指定第三个参数的值，则将由前两个参数计算出的幂对该值求余。

调用该函数时，需要按照 pow 函数的语法格式中的参数顺序指定各个参数的值。下面的代码计算 3 的 2 次幂，并将计算结果打印到屏幕上。此处将 3 传递给 base 参数，将 2 传递给 exp 参数。这种方式是按照参数的位置来指定参数的值。

```
print(pow(3, 2))
```

交换两个数字的位置将计算 2 的 3 次幂。如果希望在交换两个数字的位置后，仍然计算 3 的 2 次幂，则需要按照关键字来传递参数，此时需要使用 pow 函数的语法格式中的参数名称和等号的形式来指定参数的值。在这种情况下指定参数的顺序不重要，因为已经明确给出了参数的名称，Python 会根据名称将指定的值传递给对应的参数。

```
print(pow(exp=2, base=3))
```

需要注意的是，下面的代码将导致如图 1-6 所示的错误，这是因为按照关键字指定的参数不能出现在按照位置指定的参数之前。

```
print(pow(exp=2, 3))
```

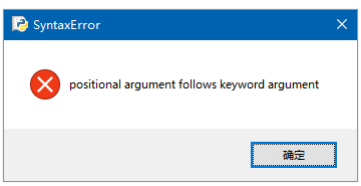

图 1-6　错误信息

pow 函数中的 mod 参数是可选的，前面的示例省略了该参数。下面的代码将该参数的值设置为 6，此时将计算 3 的 2 次幂对 6 求余后的结果，即 9 除以 6 的余数，最终结果是 3。

```
print(pow(3, 2, 6))
```

提示：为了使示例代码更简洁，减少不必要的重复，后续示例中不再添加 print 函数。但要注意的是，如果在脚本模式中运行这些示例代码，为了让代码的运行结果显示在屏幕上，必须在代码中添加 print 函数。后续示例省略 print 函数，是为了使示例中的核心代码更突出。

1.5.3　创建新的函数

在 Python 中使用 def 语句创建新的函数。创建函数时以 def 语句开头，在 def 关键字的右侧输入函数的名称，在函数名称的右侧是一对小括号，在其中输入一个或多个参数的名称，各个参数之间使用逗号分隔，def 语句以冒号结尾。创建函数时定义的参数称为形参，调用函数时为形参传递的实际数据称为实参。

def 语句是一个包含多行代码的复合语句，上面介绍的只是该语句的首行代码，用于定义函数的名称和参数。该语句的其他行代码用于定义函数的功能，这些代码行都要向右缩进指定的距离。“缩进”是 Python 中重要的语法格式，不同的缩进距离表示不同的代码层级，这也是嵌套代码的格式标准。

如果希望函数返回一个值，则使用 return 语句设置该值。省略 return 语句时函数没有返回值。下面的代码是创建一个名为 sumx 的函数，用于计算两个数字之和，并返回计算结果。

```
def sumx(x, y):
    return x + y
```

下面的代码调用 sumx 函数，并向其传递两个数字，计算结果是这两个数字的总和为 8。

```
sumx(3, 5)
```

可以使用一对三重引号为函数添加说明信息，用于介绍函数的功能和参数的数据类型等。下面的代码为前面创建的 sumx 函数添加说明信息。

```
def sumx(x, y):
    '''计算两个数字之和，两个参数必须是数字，否则将导致错误'''
    return x + y
```

编写代码时可以使用 __doc__ 属性访问函数的说明信息。下面的代码将在屏幕上打印为 sumx 函数添加的说明信息。

```
sumx.__doc__
```

1.6 面向对象编程

面向对象编程是以操作对象及其属性和方法为主的编程方式。使用面向对象的方式编写代码，可以让程序的组织结构更紧凑，代码的含义更清晰，代码的重用率更高。

1.6.1 使用 Python 内置对象

Python 中的任何东西都是对象。数字是对象，字符串也是对象，列表、元组和字典都是对象，甚至函数、模块和文件也都是对象，对象在 Python 中无处不在。Python 内置的常用对象类型如表 1-1 所示，其中的标识符是在编写代码时可以使用的表示对象类型的名称。

表 1-1 Python 内置的常用对象类型

对象类型的标识符	对象类型
int	整数
Float	浮点数
complex	复数
str	字符串
list	列表
tuple	元组
dict	字典
set	集合

在代码中输入字面值或为变量赋值的操作都会创建对象。下面的代码是创建一个整数类型的对象，它是一个字面值，即直接输入数据本身。

```
666
```

下面的代码是创建一个字符串类型的对象，将该字符串赋值给一个变量，使用该变量引用这个字符串对象。

```
name = 'sx'
```

使用 Python 内置的 type 函数可以检测数据的对象类型。下面的代码显示上面创建的 name 变量所引用的数据的对象类型：

```
type(name)
```

代码的运行结果如下，表示字符串类型。

```
<class 'str'>
```

1.6.2　使用属性获取对象的状态信息

很多对象都有与其关联的属性，使用属性可以获取对象的状态信息，正如前面为创建的函数添加说明信息时使用的 __doc__ 属性，它是函数对象的一个属性。在代码中使用对象的属性时，需要先输入对象的名称，然后输入一个英文句点，再输入属性的名称。

```
sumx.__doc__
```

提示：如果输入的对象名称正确，则在输入一个英文句点后，默认会显示一个包含属性和方法的列表，可以从中选择所需的属性，自动将其添加到当前代码中。

1.6.3　使用方法让对象执行操作

与属性类似，很多对象都有一些方法，用于执行特定的操作。例如，字符串对象的 upper 方法可以将一个字符串中的所有英文字母转换为大写。下面的代码将一个字符串赋值给一个变量，此时该变量引用的是一个字符串对象，然后在该变量上使用字符串对象的 upper 方法，将字符串中的所有英文小写字母转换为大写。

```
name = 'python'
name.upper()
```

方法也可以像函数一样包含参数。列表对象有一个 append 方法，用于在列表末尾添加一个元素，要添加的元素作为参数传递给 append 方法。下面的代码先创建一个空列表，然后在该列表中添加一个数字。

```
numbers = []
numbers.append(666)
```

方法和函数看起来非常相似，都能执行指定的操作，然而，它们的使用范围和语法格式有所不同。方法只属于特定类型的对象，所以只能在某种特定类型的对象上使用。而函数可以在多种类型的对象上使用，所以函数的通用性更强。

使用方法和函数的语法格式也有一些区别。使用方法时，需要先输入对象的名称，然后输入一个英文句点，再输入方法的名称。使用函数时，只需输入函数的名称即可。为方法和函数传递参数的方法相同。

1.6.4　创建新的对象

除了 Python 内置对象之外，用户可以创建新的对象。创建对象前需要先创建对象所属的类。可以将“类”看作是某类对象的模板，只有先定义一个类，才能基于该类创建任意数量的同类对象。将由类创建的对象称为该类的实例。

在 Python 中使用 class 语句创建新的类。创建类时以 class 语句开头，在 class 关键字的右侧输入类的名称，类的首字母通常使用大写，class 语句以冒号结尾。与创建函数使用的 def 语句类似，class 语句也是一个复合语句，除了定义类名称的首行代码之外，class 语句的其他行代码用于定义类的功能，这些代码都要向右缩进指定的距离。

在定义类功能的代码中可以包含任意数量的变量和函数，其中的变量是类的属性，函数是类的方法。下面的代码是创建一个名为 Person 的类，并为该类创建了一个属性和一个方法，属性是 name 变量，方法是 eat 函数。

```
class Person:
    name = '宋翔'
    def eat(self, food):
        print(self.name + '吃了一个' + food)
```

注意：创建类的方法时，每个方法对应的函数必须在 class 语句内部向右侧缩进，每个函数的第一行不能与 class 语句垂直对齐。在为类创建的每个方法中的第一个参数表示方法所属的对象，这个对象就是运行代码时由当前类创建的对象。在 Python 中通常使用 self 作为每个方法的第一个参数，self 并没有什么特别的含义，只不过是 Python 中的一种约定俗成。

运行上述代码后，可以使用 Person 类创建实际的对象，代码的格式与调用不带参数的函数类似。

```
p = Person()
```

创建一个对象后，可以使用以下代码获取 name 属性的值。

```
p.name
'宋翔'
```

使用对象的 eat 方法，可以在屏幕上打印对象吃的是什么食物，这个食物作为参数传递给 eat 方法。

```
p.eat('苹果')
宋翔吃了一个苹果
```

上面使用的是定义类时为 name 属性设置的默认值。也可以在创建对象后，修改 name 属性的值，以便显示指定的姓名。下面是将上面几行代码组合到一起后的代码，此处将 name 属性设置为“黄帝”。

```
p = Person()
p.name = '黄帝'
p.eat('苹果')
```

代码的运行结果如下：

```
黄帝吃了一个苹果
```

如果希望在使用类创建一个对象时，能够为对象的各个属性设置初始值，则可以在类

中定义一个名为 __init__ 的方法，其中包含想要初始化的属性。使用类创建对象时，会自动调用该类中的 __init__ 方法，并执行该方法中的代码，从而实现初始化属性值的功能。

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

下面的代码使用 Person 类创建一个对象，并在创建时以参数的方式指定 name 和 age 两个属性的初始值。

```
p = Person('黄帝', 9999)
```

使用下面两行代码可以检测 name 和 age 两个属性是否已被设置了初始值。

```
p.name
p.age
```

第 2 章

快速构建图表所需的数据

绘制图表时，NumPy 和 Pandas 两个库的主要用途是快速构建用于绘图的原始数据，Pandas 库还可以对数据进行清洗和格式化，使数据更符合图表的绘制要求。本章将介绍使用 Python 内置对象以及 NumPy 和 Pandas 中的核心对象为图表构建基础数据的方法，包括创建新的数据和从文件中读取数据两种方式。

2.1 使用 Python 中的列表对象构建数据

列表（List）对象是 Python 中最常用的对象之一，可以将一项或多项数据存储到列表中，并可以随时在列表中添加或删除数据。列表中的数据是有序排列的，每个数据都有一个索引，通过索引可以访问特定位置上的数据。

2.1.1 创建包含一项或多项数据的列表

创建一个列表时，使用一对中括号包围列表中的所有数据，各项数据之间以逗号分隔。下面的代码是创建一个包含 3 个元素的列表，这 3 个元素是一个字符串和两个数字。

```
['sx', 666, 888]
```

如果列表只包含一个元素，则只需将其放到一对中括号中。

```
['sx']
```

使用 Python 内置的 list 函数，可以将字符串或元组转换为列表。下面的代码是将字符串中的每一个字符转换为列表中的每一项数据。

```
list('python')
```

上述代码将创建以下列表：

```
['p', 'y', 't', 'h', 'o', 'n']
```

如需创建一个包含数字 1 ～ 9 的列表，可以使用以下代码，其中的 range 是一个 Python 内置函数，为 range 函数设置的参数 10 是将要创建的数字上限，但是不包括该数字。

```
list(range(1, 10))
```

上述代码将创建以下列表：

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

2.1.2　创建嵌套列表

列表中的数据可以是任何类型，如果列表中的数据也是列表，则内外列表将构成一个嵌套列表，外层列表称为父列表，内层列表称为子列表。下面的代码是创建一个嵌套列表，外层列表有两个元素，每个元素也是一个列表。内层的每个列表有 3 个元素，每个元素都是数字。

```
[[1, 2, 3], [4, 5, 6]]
```

使用 list 函数和 range 函数也可以创建上面的嵌套列表，代码如下：

```
[list(range(1, 4)), list(range(4, 7))]
```

2.1.3　创建符合特定条件的列表

如需使列表中的数据满足指定的条件，可以使用列表推导式创建列表。列表推导式是一个表达式，只需一行代码即可实现至少需要两三行代码才能实现相同功能的 for 语句，任何可以使用表达式的地方，都可以使用列表推导式，例如可以将列表推导式的结果赋值给变量。下面的代码是创建一个只包含数字 1 ～ 10 中偶数的列表。

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

上述代码将创建以下列表：

```
[2, 4, 6, 8, 10]
```

在上面的代码中，将列表推导式中的所有内容放入一对中括号中，表明将要创建一个列表。在中括号中，开头部分的 int(x) 表示将每一个数字转换为整数。从 for 开始直到结尾部分表示使用变量 x 逐一引用由 list 函数创建的列表中的每一个数字，并将能被 2 整除的数字添加到列表中。处理完列表中的最后一个数字后，列表推导式将结束运行，最终创建的就是 1 ～ 10 中的所有偶数。

2.2　使用 Python 中的元组对象构建数据

元组（Tuple）对象与列表对象有很多相似之处，可以将一项或多项数据存储到元组

中，元组中的每项数据也是有序的，也需要通过索引访问元组中的数据。与列表不同的是，一旦将数据存储到一个元组中，就不能再修改该元组中的数据了。

2.2.1 创建包含一项或多项数据的元组

如果元组只包含一项数据，则可以将该数据输入到一对小括号中，并在该数据的右侧输入一个英文逗号，即可创建只包含一项数据的元组。在这种情况下，不能省略数据末尾的逗号，否则创建的不是元组。

```
(3,)
```

创建包含多项数据元组的方法与列表类似，唯一区别是使用小括号包围各项数据或使用 tuple 函数代替 list 函数。下面的代码是创建一个包含 3 个数字的元组。

```
(1, 2, 3)
```

下面的代码使用 tuple 函数和 range 函数创建相同的 3 个数字。

```
tuple(range(1, 4))
```

使用 tuple 函数可以将字符串或列表转换为元组。下面的代码将字符串中的每一个字符转换为元组中的每一项数据。

```
tuple('python')
('p', 'y', 't', 'h', 'o', 'n')
```

2.2.2 创建符合特定条件的元组

与使用列表推导式创建列表类似，也可以使用类似的方法创建元组。将列表推导式中一对中括号替换成小括号，创建的并非元组，而是一种称为“生成器”的对象。为了创建元组，需要将表达式作为参数传递给 tuple 函数。

```
tuple(x for x in list(range(1, 11)))
```

上述代码将创建以下元组：

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

2.3 使用 Python 中的字典对象构建数据

与列表和元组不同，字典（Dict）对象中的每项数据由键和值两个部分组成。在字典中不会出现重复的键，所以可以通过唯一的键访问与该键关联的值。与列表类似，在字典

中也可以随时添加和删除数据。

2.3.1　创建包含一项或多项数据的字典

创建包含一项或多项数据的字典有以下几种方法：

- 手动输入大括号和字典中的数据。
- 使用 dict 函数将关键字参数创建为字典。
- 使用 dict 函数将序列对象转换为字典。
- 使用 dict 函数和 zip 函数创建字典。

下面分别介绍使用这几种方法创建字典。

1. 手动输入大括号和字典中的数据

创建字典最直接的方法是输入一对大括号，并在其中输入一项或多项数据，每项数据中的键和值之间以英文冒号分隔，各项数据之间以英文逗号分隔。下面的代码是创建只包含一项数据的字典，该项数据的键是“牛奶”，值是 2。

```
{'牛奶': 2}
```

下面的代码是创建包含 3 项数据的字典，每项数据由商品名称和单价组成。

```
{'牛奶': 2, '酸奶': 3, '果汁': 5}
```

2. 使用 dict 函数和关键字参数创建字典

创建字典时，手动输入每一项数据，以及引号、冒号和逗号的效率很低。使用 dict 函数能够以类似于为函数指定关键字参数的形式，将关键字参数的名称及其值转换为字典中的键和值。

下面的代码是创建与前面示例完全相同的字典，为 dict 函数指定 3 个参数，每个参数的名称被创建为字典中的键，每个参数的值被创建为字典中与键关联的值。

```
dict(牛奶=2, 酸奶=3, 果汁=5)
```

3. 使用 dict 函数将序列对象转换为字典

Python 中的字符串、列表、元组等都是序列对象，序列对象中的每项数据是有序排列的，可以被索引、切片和迭代。迭代是指程序依次处理每一项数据，直到最后一项数据为止。

使用 dict 函数可以将序列对象转换为字典，该方法要求序列对象中的每项数据都由两个值组成，第一个值被创建为字典中的键，第二个值被创建为与键关联的值。下面的代码创建与前面示例相同的字典，此处将一个列表作为参数传递给 dict 函数，该列表中的每项数据都是一个元组，每个元组都由两个值组成，第一个值是字符串，第二个值是数字。

```
dict([('牛奶', 2), ('酸奶', 3), ('果汁', 5)])
```

4. 使用 dict 函数和 zip 函数创建字典

如果字典中的键和值分别位于两个序列对象中，则可以使用 zip 函数将两个序列对象中相同位置上的值组合为一项数据，类似于上一个示例中由两个值组成的元组，然后使用 dict 函数将 zip 函数的返回值转换为字典。

下面的代码仍然是创建与前面示例相同的字典，首先创建分别表示商品名称和单价的两个列表，然后使用 zip 函数和 dict 函数将两个列表中的相关项创建为字典中每项数据的键和值。

```
names = ['牛奶', '酸奶', '果汁']
prices = [2, 3, 5]
dict(zip(names, prices))
```

2.3.2 创建符合特定条件的字典

与列表推导式类似，使用字典推导式可以创建包含满足指定条件的数据字典。下面的代码是使用变量 x 控制字典中每项数据的键和值，变量 x 的值用作每项数据的键，变量 x 的平方值用作与键关联的值，变量 x 和 x 的平方之间以英文冒号分隔。通过逐一引用由 list 函数和 range 函数构建的列表中的每一个数字来得到变量 x 的值。

```
{x: x**2 for x in list(range(1, 4))}
```

代码的运行结果如下：

```
{1: 1, 2: 4, 3: 9}
```

使用 zip 函数可以在字典推导式中使用两个变量分别控制字典中的键和值。下面的代码创建与前面示例完全相同的字典，但是此处使用的是字典推导式和 zip 函数。

```
names = ['牛奶', '酸奶', '果汁']
prices = [2, 3, 5]
{x: y for (x, y) in zip(names, prices)}
```

2.4 使用 NumPy 中的 Ndarray 对象构建数据

Ndarray 是 NumPy 的核心对象，该对象表示一维或多维数组，为了使本节中的示例代码正常运行，需要在每个示例代码的开头添加以下代码，在 Python 中导入 NumPy 库，并将其别名设置为 np，本节中的每个示例都使用该别名。

```
import numpy as np
```

2.4.1　创建一维数组

使用 NumPy 中的 arange 函数可以创建一维数组，该函数的用法与 Python 中的 range 函数类似。将一个整数作为参数传递给 arange 函数时，将创建一个从 0 到比该整数小 1 的值组成的一维数组。下面的代码创建由 0、1 和 2 三个元素组成的一维数组。

```
np.arange(3)
```

如果希望数组的第一个元素不是 0，则可以为 arange 函数同时传递两个参数，第一个参数用于指定第一个元素的值。下面的代码创建由 1 和 2 两个元素组成的一维数组。

```
np.arange(1, 3)
```

如果为 arange 函数传递第 3 个参数，则创建的数组中的各个元素之间具有指定的间隔。下面的代码创建由 1 ～ 10 的所有偶数组成的数组。

```
np.arange(2, 11, 2)
```

在交互模式中运行上面的代码，将显示以下结果：

```
array([ 2,  4,  6,  8, 10])
```

如果不想在运行结果中显示 array，则可以使用 Python 中的 print 函数，代码如下：

```
print(np.arange(2, 11, 2))
```

提示：这种问题在脚本模式中不存在，因为在脚本模式中编写要将结果显示在屏幕上的代码时，必须使用 print 函数。后面的示例将省略 print 函数，以便减少代码的复杂性。

在 NumPy 中还可以使用 array 函数将 Python 中的序列对象创建为数组。下面的代码将一个列表作为参数传递给 array 函数，将其创建为一维数组。

```
np.array([1, 2, 3])
```

传递给 array 函数的参数也可以是引用序列对象的变量。

```
numbers = [1, 2, 3]
np.array(numbers)
```

如需创建在指定数值范围内的等差数列的数组，可以使用 linspace 函数。该函数的前两个参数用于指定值的范围，第三个参数用于指定值的数量，即数组包含的元素个数。下面的代码是创建包含 3 个元素的一维数组，这 3 个元素都是 1 ～ 10 的数字，且两个相邻元素的差值是 4.5。

```
np.linspace(1, 10, 3)
```

代码的运行结果如下，任意两个相邻元素的差值都是 4.5。

```
[ 1.   5.5 10. ]
```

2.4.2 创建二维数组

使用 array 函数可以创建二维数组，只需将一个嵌套列表作为参数传给该函数即可。下面的代码创建一个二维数组，该数组的第一维包含两个元素（即两个子列表），第二维包含 3 个元素（即每个列表中的 3 个数字）。如果将该数组看作表格，则该表格有 2 行 3 列。

```
np.array([[1, 2, 3], [7, 8, 9]])
```

上述代码将创建以下数组：

```
[[1 2 3]
 [7 8 9]]
```

使用 NumPy 中的 random 函数可以创建一个包含 0 ～ 1 的随机数的二维数组，该函数位于 NumPy 库的 random 模块中。将一个元组作为参数传递给 random 函数，该元组中的值表示二维数组的行数和列数。下面的代码是创建一个 2 行 3 列的二维数组。

```
np.random.random((2, 3))
```

创建的数组如下，每次运行相同的代码会得到不同的随机数。

```
[[0.53631257 0.02745073 0.15569723]
 [0.72433551 0.36637602 0.13712715]]
```

2.4.3 将一维数组转换为二维数组

使用 Ndarray 对象的 reshape 方法，可以在使用 arange 函数创建一维数组时，直接将其转换为指定行、列数的二维数组。下面的代码将包含 6 个元素的一维数组转换为 2 行 3 列的二维数组。

```
np.arange(6).reshape(2, 3)
```

上述代码将创建以下数组：

```
[[0 1 2]
 [3 4 5]]
```

即使转换后的数组只有一行，它也是一个二维数组。

```
np.arange(6).reshape(1, 6)
```

上述代码将创建以下数组：

```
[[0 1 2 3 4 5]]
```

reshape 方法不会改变原数组的维数，当将创建的数组赋值给变量时，然后对该变量

使用 reshape 方法转换数组时，即可证实这种情况。

如需修改原数组的维数，可以使用 Ndarray 对象的 shape 属性。下面的代码是实现与前面示例相同的功能，但是使用的是 shape 属性，将一个表示数组行、列数的元组赋值给该属性。

```
np.arange(6).shape = (2, 3)
```

或

```
np.arange(6).shape = 2, 3
```

2.4.4　查看数组的维数和元素数

使用 Ndarray 对象的 ndim 属性，可以查看数组的维数。下面的代码是先创建一个二维数组，然后使用 ndim 属性查看该数组的维数，该属性的返回值是 2，表示该数组有两个维度，是一个二维数组。

```
numbers = np.arange(6).reshape(2, 3)
numbers.ndim
```

使用 Ndarray 对象的 shape 属性，可以查看数组的形状，即数组每个维度包含的元素数。下面的代码是查看上面创建的 numbers 数组的形状：

```
numbers.shape
```

由于 numbers 数组有 2 行 3 列，所以将以元组的形式返回以下结果：

```
(2, 3)
```

如需查看数组包含的元素总数，可以使用 Ndarray 对象的 size 属性。下面的代码是查看 numbers 数组包含的元素总数，由于该数组有 2 行 3 列，所以元素总数是 2×3=6。

```
numbers.size
```

2.4.5　修改数组元素的值

与修改 Python 列表中元素的值的方法类似，如需修改 NumPy 数组中元素的值。可以将该元素的索引放到数组名右侧的一对中括号中，然后在等号的右侧输入修改后的值。下面的代码是先创建包含 0、1 和 2 三个元素的数组，然后将该数组的第 2 个元素的值修改为 666。

```
numbers = np.arange(3)
numbers[1] = 666
```

代码的运行结果如下，由于第 2 个元素是 3 位数，所以 NumPy 会自动使用空格将该数组中的其他元素补足到 3 位。

```
[  0 666   2]
```

注意：数组中所有元素的数据类型都必须相同，修改元素时，如果为其赋值的数据类型与其他元素的数据类型不同，则将导致错误。

可以一次性修改多个元素的值，此时需要将这些元素的索引添加到一个列表中，然后使用该列表对数组进行索引。下面的代码是将数组中的第 1 个元素和第 3 个元素的值分别修改为 666 和 888。也可以使用一个变量引用列表，然后在数组名右侧的中括号中使用变量名代替列表。

```
numbers = np.arange(3)
numbers[[0, 2]] = 666, 888
```

修改后的数组如下：

```
[666   1 888]
```

2.4.6 转置数组的行和列

使用 Ndarray 对象的 T 属性，可以将数组的行转换为列，将数组的列转换为行。下面的代码是创建一个 2 行 3 列的数组。

```
numbers = np.arange(6).reshape(2, 3)
[[0 1 2]
 [3 4 5]]
```

使用 T 属性将该数组转换为 3 行 2 列。

```
numbers.T
[[0 3]
 [1 4]
 [2 5]]
```

使用 Ndarray 对象的 transpose 函数也可以实现相同的功能。

```
np.transpose(numbers)
```

以上两种方法都不会将原数组修改为转置后的数组。

2.5 使用 Pandas 中的 Series 对象构建数据

Series 是 Pandas 的核心对象之一，该对象与 Python 中的列表对象类似，它们存储数

据的方式与一维数组类似。不过 Series 对象还包含与数据对应的索引（标签）。为了使本节的示例代码正常运行，需要在每个示例的开头添加以下代码，在 Python 中导入 Pandas 库，并将其别名设置为 pd，本节中的每个示例都使用该别名。

```
import pandas as pd
```

2.5.1　创建 Series 对象

使用 Pandas 中的 Series 函数可以创建 Series 对象，该函数主要有以下两个参数：

- data：存储在 Series 对象中的数据，可以是 Python 中的数字、字符串、列表、元组和字典，也可以是 NumPy 中的一维数组。
- index：与 data 参数指定的各项数据对应的索引。省略该参数时，自动创建从 0 开始的整数序列，并将其用作数据的索引。

1. 使用 Python 列表创建 Series 对象

下面的代码是使用 Python 中的列表对象作为 data 参数来创建 Series 对象，并将从 1 开始的自然数序列设置为数据的索引，最后一个索引编号会根据列表中的元素总数自动调整。

```
data = list('python')
index = list(range(1, len(data)+1))
sr = pd.Series(data, index)
```

上述代码将创建以下数据：

```
1   p
2   y
3   t
4   h
5   o
6   n
```

为 index 变量赋值时，可以省略 list 函数，将上面的代码改为以下形式：

```
index = range(1, len(data)+1))
```

如果不想使用中间变量 data 和 index，则可以将 3 行代码合并为一行：

```
sr = pd.Series(list('python'), range(1, len(names)+1))
```

如果省略 index 参数，则创建的 Series 对象中的数据起始索引将从 0 开始。

```
0   p
1   y
2   t
```

```
3   h
4   o
5   n
```

除了数字之外，也可以使用字符串作为索引。下面的代码是使用从 A 开始的英文大写字母作为数据的索引。

```
data = list('python')
index = list('ABCDEF')
sr = pd.Series(data, index)
```

上述代码将创建以下数据：

```
A   p
B   y
C   t
D   h
E   o
F   n
```

data 参数的值也可以是手动输入元素的列表，列表中的各个元素可以具有不同的数据类型。

```
data = [1, 2, 3, 'Python', 'NumPy', 'Pandas']
index = list('ABCDEF')
sr = pd.Series(data, index)
```

上述代码将创建以下数据：

```
A         1
B         2
C         3
D    Python
E     NumPy
F    Pandas
```

提示：当 index 参数中的索引个数大于 data 参数中的数据个数时，多出的索引所对应的数据将显示为 NaN（Not a Number），这是在 Pandas 中表示缺失数据的方法。

2. 使用 Python 字典创建 Series 对象

下面的代码与上一个示例的功能相同，此处使用 Python 中的字典对象作为 data 参数来创建 Series 对象，此时会将字典中的键作为数据的索引，所以无须额外指定 index 参数。

```
data = dict(zip(range(1, 7), list('python')))
sr = pd.Series(data)
```

如果在使用字典的情况下仍然指定了 index 参数，则最后创建的数据的索引将按照

index 参数的顺序显示，并从字典中提取相应的值。

```
data = dict(zip(range(1, 7), list('python')))
sr = pd.Series(data, [1, 3, 5, 2, 4, 6])
```

上述代码将创建以下数据：

```
1   p
3   t
5   o
2   y
4   h
6   n
```

3. 使用字面值创建 Series 对象

下面的代码使用字面值作为 data 参数来创建 Series 对象。由于 index 参数有 6 个值（1 ～ 6），而 data 参数只有一个值（666），为了使每个索引都有对应的值，所以会将 666 重复显示 6 次，使每个索引都有一个值。

```
sr = pd.Series(666, range(1, 7))
```

上述代码将创建以下数据：

```
1   666
2   666
3   666
4   666
5   666
6   666
```

提示：Python 中的字面值在 Pandas 和 NumPy 中称为标量值，表示没有方向的值。

4. 使用 NumPy 数组创建 Series 对象

下面的代码是使用 NumPy 中的 arange 函数创建一个一维数组，并将其作为 data 参数来创建 Series 对象。

```
sr = pd.Series(np.arange(100, 106))
```

上述代码将创建以下数据：

```
0   100
1   101
2   102
3   103
4   104
5   105
```

2.5.2 获取或修改 Series 对象中的数据

与在 Python 字典中通过键来获取值的方法类似，在 Pandas 中可以使用索引获取 Series 对象中与索引关联的数据。下面的代码所创建的 Series 对象包含 6 个英文字母，使用索引 0 和 5 显示第 1 个和第 6 个英文字母。

```
sr = pd.Series(list('pandas'))
sr[0]
sr[5]
```

代码的运行结果如下：

```
'p'
's'
```

使用类似的方法，还可以修改 Series 对象中的数据。下面的代码将第二个英文字母修改为“y”。

```
sr[1] = 'y'
```

利用切片，可以一次性修改 Series 对象中的多项数据。下面的代码将第 3 ～ 6 个英文字母修改为“thon”。本例中冒号左侧的数字表示索引的起始编号，冒号右侧留空表示直到数据结尾。如果将冒号左侧留空，则表示“从数据开头开始”。

```
sr[2:] = list('thon')
```

注意：使用无效的索引将导致错误。为了避免这种情况，可以使用 Series 对象的 get 方法，检测到无效索引时该方法不会返回任何值，或者可以为其指定在遇到无效索引时显示哪些信息。下面的代码在使用无效索引时显示“不存在该索引”。

```
sr.get(6, '不存在该索引')
'不存在该索引'
```

2.5.3 为 Series 对象命名

为 Series 对象命名有以下几种方法：

- 创建 Series 对象时，使用 name 关键字参数。
- 创建 Series 对象后，使用该对象的 name 属性。
- 创建 Series 对象后，使用该对象的 rename 方法。

下面的代码使用第 1 种方法将 Series 对象命名为 py。使用 Series 对象的 name 属性可以获取该对象的名称。

```
sr = pd.Series(list('python'), name='py')
sr.name
```

代码的运行结果如下：

```
'py'
```

下面的代码使用第 2 种方法将 Series 对象命名为 py。

```
sr = pd.Series(list('python'))
sr.name = 'py'
```

下面的代码使用第 3 种方法将 Series 对象命名为 py。

```
sr = pd.Series(list('python'))
sr.rename('py')
```

前两种方法的功能相同，命名后都可以使用 Series 对象的 name 属性获取名称。第 3 种方法不会将名称保存到 name 属性中，所以使用 name 属性无法获取名称。

2.5.4　获取 Series 对象中的所有数据

使用 Series 对象的 array 属性可以获取该对象中的所有数据。

```
sr = pd.Series(list('python'))
sr.array
```

返回的 Series 对象中的数据如下，第一行显示的是数据类型，array 属性返回的数据类型是 Pandas 内部定义的 NumPy 扩展数组。

```
<NumpyExtensionArray>
['p', 'y', 't', 'h', 'o', 'n']
Length: 6, dtype: object
```

如需返回 NumPy 中的 Ndarray 对象数据类型，可以使用 Series 对象的 to_numpy 方法。

```
sr = pd.Series(list('python'))
sr.to_numpy()
```

返回的 Series 对象中的数据如下：

```
['p' 'y' 't' 'h' 'o' 'n']
```

使用 Python 中的 type 函数可以检测 to_numpy 方法返回值的数据类型。

```
type(sr.to_numpy())
```

返回的数据类型如下：

```
<class 'numpy.ndarray'>
```

2.6 使用 Pandas 中的 DataFrame 对象构建数据

DataFrame 是 Pandas 的另一个核心对象，该对象包含一组有序的列，每列可以是不同类型的数据。当 DataFrame 对象包含多列数据时，其结构类似于 Excel 工作表或数据库应用程序中的表。为了便于标识特定的行、列或行列交错位置上的值，DataFrame 对象同时包含行索引和列索引。

与 Series 对象类似，创建 DataFrame 对象也可以使用多种类型的数据。

- Python 中的列表对象。
- Python 中的字典对象。
- NumPy 中的 Ndarray 对象，可以是一维数组或二维数组。
- Pandas 中的 Series 对象。

使用 Pandas 中的 DataFrame 函数可以创建 DataFrame 对象，该函数主要有以下 3 个参数：

- data：存储在 DataFrame 对象中的数据。
- index：行索引。
- columns：列索引。

与创建 Series 对象类似，使用 DataFrame 对象构建数据时，也需要在 Python 中导入 Pandas 库。

2.6.1 使用 Python 中的列表对象创建 DataFrame 对象

下面的代码是使用 Python 中的列表对象作为 data 参数来创建 DataFrame 对象，由于未指定 DataFrame 对象的行索引和列索引，所以默认使用从 0 开始的两组整数序列作为行索引和列索引。本例中的列表包含 6 个字母，每个字母都是列表中的一个元素，使用该数据创建的 DataFrame 对象包含一列数据，该列数据有 6 行。由于没有指定 index 参数，所以列索引为 0，行索引为 0 ～ 5。

```
data = list('python')
df = pd.DataFrame(data)
```

上述代码将创建以下数据：

```
   0
0  p
1  y
2  t
3  h
4  o
5  n
```

如需创建两列数据，则可以构建一个嵌套列表。下面代码运行的结果可能出乎意料，两个子列表中的数据并没有像上一个示例那样纵向排列，而是横向排列。

```
data = [list('python'), list('pandas')]
df = pd.DataFrame(data)
```

上述代码将创建以下数据：

```
   0  1  2  3  4  5
0  p  y  t  h  o  n
1  p  a  n  d  a  s
```

可以使用 NumPy 中的 transpose 函数转置数据的行和列，修改后的代码如下：

```
data = [list('python'), list('pandas')]
df = pd.DataFrame(np.transpose(data))
```

上述代码将创建以下数据：

```
   0  1
0  p  p
1  y  a
2  t  n
3  h  d
4  o  a
5  n  s
```

如需使行索引和列索引都从 1 开始编号，可以在创建 DataFrame 对象时指定 index 和 columns 两个参数。

```
data = [list('python'), list('pandas')]
index = range(1, 7)
columns = range(1, 3)
df = pd.DataFrame(np.transpose(data), index, columns)
```

上述代码将创建以下数据：

```
   1  2
1  p  p
2  y  a
3  t  n
4  h  d
5  o  a
6  n  s
```

下面的代码可以让行索引和列索引的最大值随着数据范围的大小自动调整，其中的 data[0] 引用第一个子列表。由于两个子列表中的元素个数相同，所以引用哪一个子列表都可以。

```
index = range(1, len(data[0])+1)
column = range(1, len(data)+1)
```

2.6.2 使用 Python 中的字典对象创建 DataFrame 对象

使用 Python 中的字典对象作为 data 参数来创建 DataFrame 对象时，会自动将字典的键作为 DataFrame 对象的列索引。下面的代码将创建一个键为 1 和 2 的字典，与每个键关联的值都是一个列表。创建 DataFrame 对象后，使用这两个键作为两列数据的列索引。可以发现，使用这种方法创建的 DataFrame 对象，无须对其中的两组数据的行和列进行转置。

```
data = {1: list('python'), 2: list('pandas')}
df = pd.DataFrame(data)
```

上述代码将创建以下数据：

```
   1  2
0  p  p
1  y  a
2  t  n
3  h  d
4  o  a
5  n  s
```

2.6.3 使用 NumPy 中的 Ndarray 对象创建 DataFrame 对象

下面的代码是使用 NumPy 中的 Ndarray 对象作为 data 参数来创建 DataFrame 对象。由于使用 NumPy 中的 array 函数创建的数组为 2 行 6 列，所以需要使用 Ndarray 对象的 T 属性将该数组转换为 6 行 2 列，然后为其添加从 1 开始编号的行索引和列索引。

```
data = np.array([list('python'), list('pandas')])
df = pd.DataFrame(data.T, range(1, 7), [1, 2])
```

上述代码将创建以下数据：

```
   1  2
1  p  p
2  y  a
3  t  n
4  h  d
5  o  a
6  n  s
```

2.6.4　使用 Pandas 中的 Series 对象创建 DataFrame 对象

由于 Series 对象本身可以设置索引，所以使用该对象作为 data 参数来创建 DataFrame 对象时，会自动将 Series 对象自身的索引作为 DataFrame 对象的行索引。下面的代码是创建单列数据，其行索引就是为 Series 对象设置的索引。

```
sr = pd.Series(list('python'), range(1, 7))
df = pd.DataFrame(sr)
```

上述代码将创建以下数据：

```
   0
1  p
2  y
3  t
4  h
5  o
6  n
```

如需创建两列数据，可以创建两个 Series 对象，然后将它们组合在一个字典对象中，并为它们分别提供一个键，这些键将在创建的 DataFrame 对象中被用作列索引。

```
sr1 = pd.Series(list('python'), range(1, 7))
sr2 = pd.Series(list('pandas'), range(1, 7))
df = pd.DataFrame({1: sr1, 2: sr2})
```

上述代码将创建以下数据：

```
   1  2
1  p  p
2  y  a
3  t  n
4  h  d
5  o  a
6  n  s
```

如果创建的 DataFrame 对象包含多列，则可以使用 dict 函数和 zip 函数组合多个键和列表。对于本例来说，可以将第 3 行代码改为以下形式。当包含多列数据时，使用这种方式的输入效率更高。

```
df = pd.DataFrame(dict(zip((1, 2), [sr1, sr2])))
```

2.6.5　获取指定的行、列和值

从 DataFrame 对象中获取不同范围的数据有以下几种方法：

- 获取指定的列：使用索引或键的方式，格式为 df[列索引]。
- 获取指定的行：使用 DataFrame 对象的 loc 方法和行索引，格式为 df.loc[行索引]。还可以使用 iloc 方法和行的位置索引，例如 df.iloc[0] 表示获取第一行，将 0 改为 1 表示获取第二行。位置索引与指定的行索引无关，无论将行索引指定为何种内容，位置索引都从 0 开始编号。
- 获取指定的值：使用 DataFrame 对象的 loc 方法并指定行索引和列索引，格式为 df.loc[行索引 , 列索引]。

2.7 使用 Python 和 Pandas 读取文件中的数据

进行可视化设计的数据通常都来自于特定类型的文件，最常见的文件类型是文本文件、CSV 文件和 Excel 文件。本节将介绍使用 Python 内置功能和 Pandas 库中的功能来读取这些文件中的数据，为数据的可视化设计做好准备。

2.7.1 使用 Python 内置功能读取文本文件中的数据

文本文件的扩展名是 .txt，它是一种跨平台的通用文件格式，适合在不同的操作系统和程序之间交换数据。使用 Python 内置的 open 函数可以打开文本文件，然后使用几种方法读取其中的数据。图 2-1 是本小节要读取的文本文件。

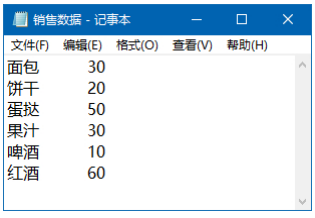

图 2-1 要读取的文本文件

读取一个文本文件之前，需要先使用 Python 内置的 open 函数打开这个文件。open 函数的第一个参数表示文件的路径，第二个参数表示文件的打开模式。由于本书只涉及读取文件的操作，所以只需将第二个参数设置为 r，该字母表示以读取模式打开文本文件。下面的代码将打开名为“销售数据 .txt”的文本文件。

```
open('E:\\ 测试数据 \\Python\\ 销售数据 .txt', 'r')
```

如果没有指定文件的路径，则假定该文件位于当前目录中。如果当前目录中不存在该文件，则将导致错误，在 Python 中将错误称为异常。

读取文件中的数据后，应该使用文件对象的 close 方法关闭该文件，以便释放其所占

用的系统资源。通常将打开的文件赋值给一个变量，该变量将表示一个文件对象，关闭文件时使用该变量调用 close 方法。

下面的代码将打开的文本文件赋值给 file 变量，然后使用该变量表示的文件对象调用 close 方法来关闭该文件。

```
file = open('E:\\ 测试数据 \\Python\\ 销售数据 .txt', 'r')
file.close()
```

文件对象提供了几种读取文本文件的方法：

- readline：每次读取一行，返回该行文本的字符串。
- readlines：一次性读取所有行，返回由各行文本组成的列表。
- read：一次性读取所有文本，返回由所有文本组成的字符串。

在对数据进行可视化设计时，通常需要读取所有文本。readlines 和 read 都用于读取文件中的所有文本，前者以行为单位进行读取，并返回一个由各行文本组成的列表对象，后者直接读取所有文本并返回一个字符串。此处只介绍 readlines 方法，因为该方法返回的内容更便于处理成适合可视化设计的数据。

使用 readlines 方法将一次性读取文本文件中的所有行，并返回一个由各行文本组成的列表，每一行文本都是列表中的一个元素。下面的代码将读取“销售数据 .txt”文件中的所有行，并将读取到的文本打印出来。

```
file = open('E:\\ 测试数据 \\Python\\ 销售数据 .txt', 'r')
data = file.readlines()
print(data)
```

代码的运行结果如下：

```
['面包 \t30\n', '饼干 \t20\n', '蛋挞 \t50\n', '果汁 \t30\n', '啤酒 \t10\n', '红酒 \t60\n']
```

在每个元素中都包含“\t”和“\n”，“\t”表示制表符，“\n”表示每行文本结尾的换行符。如需去除每个元素中的“\n”，可以使用列表推导式，在其中使用字符串对象的 rstrip 方法删除字符串结尾的字符。虽然“\n”由两个字符组成，但 Python 会将其当做单个字符处理。

```
file = open('E:\\ 测试数据 \\Python\\ 销售数据 .txt', 'r')
data = [s.rstrip() for t in file.readlines()]
print(data)
```

代码的运行结果如下：

```
['面包 \t30', '饼干 \t20', '蛋挞 \t50', '果汁 \t30', '啤酒 \t10', '红酒 \t60']
```

还可以使用字符串对象的 replace 方法删除每个元素中的“\n”，代码如下。replace 的优势并非删除换行符，而是可以替换或删除任意指定的字符。

```
data = [s.replace('\n', '') for s in file.readlines()]
```

从文件中读取到的数据通常包含多列，但是在使用诸如 Matplotlib 库为数据创建图表时，需要使用多列数据中的某一列，此时可以使用下面的代码将读取到的多列数据拆分为多个单列。

```
file = open('E:\\测试数据\\Python\\销售数据.txt', 'r')
names = []
counts = []
for line in file.readlines():
    names.append(line.rstrip().split('\t')[0])
    counts.append(int(line.rstrip().split('\t')[1]))
file.close()
```

在上面的代码中添加下面的代码，可以检测两个变量中保存的数据。

```
print(names, counts, sep='\n')
```

代码的运行结果如下，此时已将读取到的两列数据拆分为两个单列数据。

```
['面包', '饼干', '蛋挞', '果汁', '啤酒', '红酒']
['30', '20', '50', '30', '10', '60']
```

在上面的代码中，下面的表达式返回的是读取到的每个元素中的第一个部分，即商品的名称。line.rstrip() 返回去除换行符后的内容，然后使用 split 方法以制表符作为分隔符，将内容拆分为两个部分，[0] 表示提取拆分后的第一个部分，即商品的名称。[1] 表示提取拆分后的第二个部分，即商品的数量。

```
line.rstrip().split('\t')[0]
```

2.7.2 使用 Python 标准库模块读取 CSV 文件中的数据

CSV 文件是一种使用特定符号分隔各列数据的文本文件，通常以逗号分隔，但是也可以使用其他符号分隔。使用 Python 标准库中的 csv 模块可以读取 CSV 文件中的数据。图 2-2 是本小节要读取的 CSV 文件。

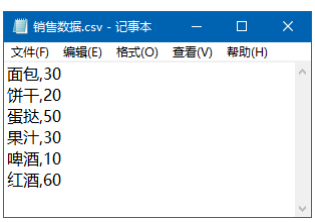

图 2-2 要读取的 CSV 文件

虽然 csv 是 Python 标准库中的模块，但是在使用该模块之前，仍然需要像使用第

三方模块一样，使用 import 语句导入模块。然后可以使用 csv 模块中的 reader 函数读取 CSV 文件中的数据，该函数的返回值是一个 reader 对象。

下面的代码是使用 open 函数以读取模式打开一个 csv 文件，然后使用该模块中的 reader 函数读取 csv 文件中的所有数据。如需在屏幕上打印读取到的数据，需要将 reader 函数的返回值作为参数传递给 list 函数。

```
import csv
file = open('E:\\测试数据\\Python\\销售数据.csv', 'r')
data = csv.reader(file)
print(list(data))
file.close()
```

代码的运行结果如下：

```
[['面包', '30'], ['饼干', '20'], ['蛋挞', '50'], ['果汁', '30'], ['啤酒', '10'], ['红酒', '60']]
```

如需将读取到的多列数据拆分为多个单列数据，可以使用下面的代码：

```
import csv
file = open('E:\\测试数据\\Python\\销售数据.csv', 'r')
names = []
counts = []
for x, y in csv.reader(file):
    names.append(x)
    counts.append(int(y))
file.close()
```

在上面的代码中添加下面的代码，可以检测两个变量中保存的数据。由于本例 CSV 文件中的数据只有两列，所以在 for 语句中使用 x 和 y 两个变量。使用的变量数量由文件中的数据总列数决定。

```
print(names, counts, sep='\n')
```

代码的运行结果如下：

```
['面包', '饼干', '蛋挞', '果汁', '啤酒', '红酒']
[30, 20, 50, 30, 10, 60]
```

2.7.3　使用 Pandas 库读取文本文件中的数据

使用 Pandas 库中的 read_table 函数，可以读取文本文件中的数据，该函数将返回 Pandas 中的 DataFrame 对象。使用 read_table 函数时最常用的有以下几个参数：

- filepath：文本文件的路径，必须指定该参数。除了该参数之外，其他参数都是关

键字参数。

- sep：各列数据之间的分隔符，未指定该参数时，其值默认为制表符。
- header：如果文本文件中的格列没有标题，则需要将该参数设置为 None。
- names：为读取后的各列数据添加列标题。
- encoding：如果文本文件中包含中文，则需要将该参数设置为一种中文编码方式，通常将其设置为 gb2312。

下面的代码将读取 2.7.1 小节中名为“销售数据 .txt”的文本文件，但是此处使用的是 Pandas 库中的 read_table 函数。由于该文件不包含列标题，所以需要将 header 参数设置为 None。

```
import pandas as pd
file = 'E:\\ 测试数据 \\Python\\ 销售数据 .txt'
df = pd.read_table(file, header=None, encoding='gb2312')
print(df)
```

代码的运行结果如下：

```
   0    1
0  面包  30
1  饼干  20
2  蛋挞  50
3  果汁  30
4  啤酒  10
5  红酒  60
```

如需为读取后的数据添加自定义的列标题，可以将列标题以列表的形式设置为 names 参数的值，并省略 header 参数，修改后的代码如下：

```
import pandas as pd
file = 'E:\\ 测试数据 \\Python\\ 销售数据 .txt'
names = [' 名称 ', ' 数量 ']
df = pd.read_table(file, names=names, encoding='gb2312')
print(df)
```

代码的运行结果如下：

```
   名称  数量
0  面包  30
1  饼干  20
2  蛋挞  50
3  果汁  30
4  啤酒  10
5  红酒  60
```

2.7.4　使用 Pandas 库读取 CSV 文件中的数据

使用 Pandas 库中的 read_csv 函数可以读取 CSV 文件中的数据，该函数的语法和用法与 2.7.3 小节介绍过的 read_table 函数基本相同，主要区别是在省略 sep 参数时，read_csv 函数默认读取以逗号分隔的数据，而 read_table 函数默认读取以制表符分隔的数据。

下面的代码将读取与 2.7.2 小节相同的 CSV 文件，并为读取后的数据添加列标题。

```
import pandas as pd
file = 'E:\\ 测试数据 \\Python\\ 销售数据 .csv'
columns = [' 名称 ', ' 数量 ']
df = pd.read_csv(file, names=columns, encoding='gb2312')
print(df)
```

2.7.5　使用 Pandas 库读取 Excel 文件中的数据

使用 Pandas 库中的 read_excel 函数可以读取 Excel 文件的数据，该函数的大多数参数都与前面介绍的 read_tableread_csv 两个函数相同，不过 read_excel 函数也有一些特殊的参数专门用于处理 Excel 文件。

下面的代码将读取名为“销售数据 .xlsx”的文件中位于第一个工作表中的数据，该文件中的数据如图 2-3 所示。

```
import pandas as pd
file = 'E:\\ 测试数据 \\Python\\ 销售数据 .xlsx'
df = pd.read_excel(file)
print(df)
```

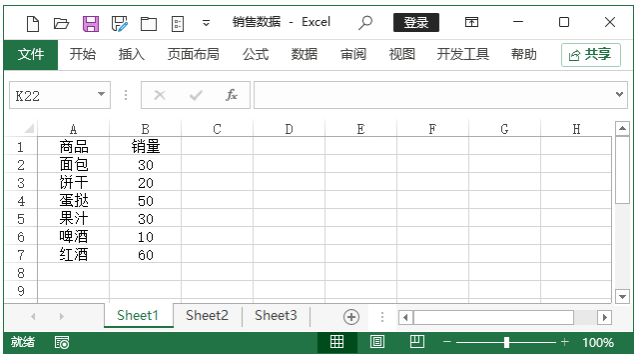

图 2-3　要读取的 Excel 文件中的数据

代码的运行结果如下：

```
   商品  销量
```

```
0  面包  30
1  饼干  20
2  蛋挞  50
3  果汁  30
4  啤酒  10
5  红酒  60
```

由于本例的 Excel 数据包含标题行，所以读取后会显示各列的标题。如果 Excel 数据不包含标题行，或者想要使用自定义标题替换原有标题，则可以为 read_excel 函数指定 names 参数。下面的代码将使用字母 A 和 B 作为两列数据的标题。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, names=['A', 'B'])
print(df)
```

代码的运行结果如下：

```
    A    B
0  面包  30
1  饼干  20
2  蛋挞  50
3  果汁  30
4  啤酒  10
5  红酒  60
```

如需读取特定工作表中的数据，可以为 read_excel 函数指定 sheet_name 参数，将该参数的值设置为一个或多个工作表的名称或索引，第一个工作表的索引为 0，其他工作表的索引以此类推。下面的代码将读取名为 Sheet2 工作表中的数据。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, sheet_name='Sheet2')
print(df)
```

下面的代码将读取 Sheet1 和 Sheet3 两个工作表中的数据。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, sheet_name=['Sheet1', 'Sheet3'])
print(df)
```

下面的代码将读取第 2 个和第 3 个工作表中的数据。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, sheet_name=[1, 2])
```

```
print(df)
```

有时可能只想处理 Excel 工作表中某些列数据，而非所有列，为了提高处理效率，可以为 read_excel 函数指定 usecols 参数，将该参数的值设置为列字母或列的索引，列的索引从 0 开始编号。下面的代码将读取 Sheet1 工作表中的 A 列数据。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, sheet_name='Sheet1', usecols='A')
print(df)
```

下面的代码将读取 Sheet1 工作表中的 A、B、C 三列数据。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, sheet_name='Sheet1', usecols='A:C')
print(df)
```

下面的代码将读取 Sheet1 工作表中的第 1、3、5 列数据。

```
import pandas as pd
file = 'E:\\测试数据\\Python\\销售数据.xlsx'
df = pd.read_excel(file, sheet_name='Sheet1', usecols=[0, 2, 4])
print(df)
```