云南大学数学与统计学院

上机实践报告

|  |  |  |
| --- | --- | --- |
| **课程名称**：数据结构与算法实验 | **年级**：2015级 | **上机实践成绩**： |
| **指导教师**：陆正福 | **姓名**：刘鹏 |  |
| **上机实践名称**：高级语言基本编程实验 | **学号**：20151910042 | **上机实践日期**：2017-05-14 |
| **上机实践编号**：No.01 | **组号**： | **上机实践时间**：上午3、4节 |

# 一、实验目的

1. 熟悉基本的Python编程，为数据结构与算法的学习奠定实验基础

2. 熟悉教材第一章的代码片段

3. 与其它程序设计语言（如C/C++/Java语言等）作对比。

# 二、实验内容

1. Python程序的编辑、编译、运行（建议使用IDLE）

2. 主讲教材第一章的Python程序的调试

3. 其它集成开发环境(IDE)的安装、配置、使用（选做），在熟悉基本操作后，可以转入其它集成开发环境的使用，如可选用Eclipse。

# 三、实验平台

Windows 10 1703 Enterprise 中文版；

Python 3.6.0；

Wing IDE Professional 6.0.5-1集成开发环境。

# 四、实验记录与实验结果分析

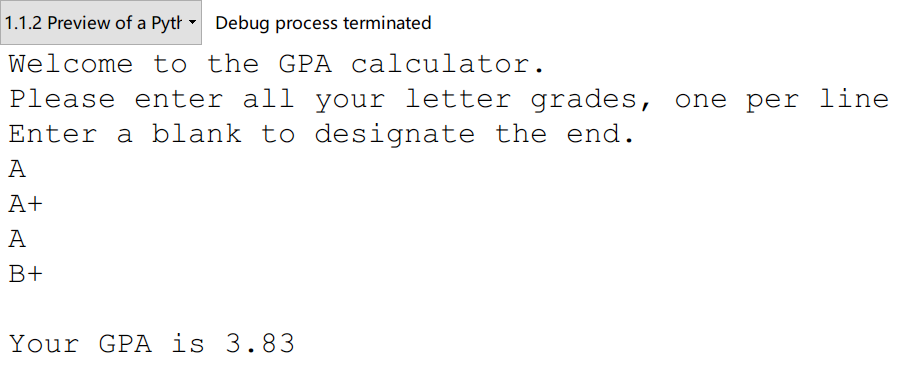
1题

程序代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | # 1.1.2 Preview of a Python Program  **print(**'Welcome to the GPA calculator.'**)**  **print(**'Please enter all your letter grades, one per line'**)**  **print(**'Enter a blank to designate the end.'**)**  # map from letter grade to point value  points **=** **{**'A+'**:**4.0**,**'A'**:**4.0**,**'A-'**:**3.67**,**'B+'**:**3.33**,**'B'**:**3.0**,**'B-'**:**2.67**,**\  'C+'**:**2.33**,**'C'**:**2.0**,**'C-'**:**1.67**,**'D+'**:**1.33**,**'D'**:**1.0**,**'F'**:**0.00**}**  num\_courses **=** 0  total\_points **=** 0  done **=** **False**  **while** **not** done**:**  grade **=** input**()** # read line from user  **if** grade **==** ''**:** # empty line was entered  done **=** **True**  **elif** grade **not** **in** points**:** # unrecognized grade entered  **print(**"Unknow grade '{0}' being ignored"**.**format**(**grade**))**  **else:**  num\_courses **+=** 1  total\_points **+=** points**[**grade**]**  **if** num\_courses **>** 0**:** # avoid division by zero  **print(**'Your GPA is {0:.3}'**.**format**(**total\_points **/** num\_courses**))** |

程序代码 1

输出结果



运行结果 1

代码分析：

从代码本身来看，可以看出这一段简单代码中包含了很多Python的基本操作，比如基本的循环控制，判断，而且相比较于曾经学过的C语言，Python还多了字典这个类型。如果要用C语言实现字典功能，可能就要写一个头文件进行预先定义，从语言的应用角度看，功能越丰富的工具更称手，而且更能使人专注于自己的主要目标。

# 五、教材翻译

**Translation**

**Chapter 1 Python Primer**

＊第一章 Python语言入门

1.1Python Overview

＊1.1Python概览

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communications is using a high-level computer language, such as Python. The Python programming language was originally developed by Guido van Rossum in the early 1990s, and has since become a prominently used language in industry and education. The second major version of the language, Python 2, was released in 2000, and the third major version, Python 3, released in 2008. We note that there are significant incompatibilities between Python 2 and Python 3. *This book is based on Python 3 (more specifically, Python 3.1 or later)*. The latest version of the language is freely available at www.python.org, along with documentation and tutorials.

＊构建数据结构以及算法的时候，我们需要给计算机下达详细的指令。要想实现这种人机对话，可采取的方式就是采用一门高级计算语言，比如说Python。Python这门编程语言是由Guido van Rossum在上个世纪九十年代早期建立，而且自此以后成为了一门被广泛用于工业与教育方面的语言。Python的第二个主要版本，即Python 2，已经于2000年发布，而它的第三个版本也于2008年发布，这也就是Python 3。我们将会注意到两个版本Python有着明显的不兼容。这本书是基于Python 3 （Python 3.1及以后的版本）编著的。Python 3 的最新版本可以在[www.python.org](http://www.python.org)进行免费下载，同时这个网站还提供相应文档与教程。

In this chapter, we provided an overview of the Python programming language and we continue this discussion in the next chapter, focusing on object-oriented principles. We assume that readers of this book have prior programming experience, although not necessarily using Python. This book does not provide a complete description of the Python language (there are numerous language references for that purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

＊在这一章中，我们将给出一些有关Python语言的概览，而且我们将会在接下来的一章中重点介绍面向对象原则。我们假定这本书的读者已经有了前期编程经验，当然这不一定必须是Python方面的。这本书没有给出有关Python语言的完整描述（有关方面的参考是浩如烟海），但是在这本书中将要用到的那些语言知识，我们都会给出介绍。

1.1.1 The Python Interpreter

＊1.1.1节 Python

Python is formally an ***interpreted*** language. Commands are executed through a piece of software known as the ***Python interpreter***. The interpreter receives a command, evaluates that command, and reports the result of the command. While the interpreter can be used interactively (especially when debugging), a programmer typically deﬁnes a series of commands in advance and saves those commands in a plain text ﬁle known as ***source code*** or a ***script***. For Python, source code is conventionally stored in a ﬁle named with the .py sufﬁx (e.g., demo.py).

＊Python是解释性语言。命令通过Python解释器执行。解释器接收命令，评估该命令，并返回命令的结果。虽然解释器可以交互式地使用（特别是在调试时），但程序员通常会提前定义一系列命令，并将这些命令保存在一个被称为源代码或脚本的纯文本文件中。对于Python而言，源代码通常存储在后缀是.py的文件里（例如，demo.py）。

On most operating systems, the Python interpreter can be started by typing python from the command line. By default, the interpreter starts in interactive mode with a clean workspace. Commands from a predeﬁned script saved in a ﬁle (e.g., demo.py) are executed by invoking the interpreter with the ﬁlename as an argument (e.g., python demo.py), or using an additional -i ﬂag in order to execute a script and then enter interactive mode (e.g., python -i demo.py).

＊在大多数操作系统上，Python解释器可以通过从命令行键入python来启动。默认情况下，解释器从一个空白的工作区启动。通过使用文件名作为参数（例如，python demo.py）调用解释器来执行保存在文件（例如，demo.py）中的命令，或使用-i附加参数执行脚本，然后进入交互模式（例如，python -i demo.py）。

Many ***integrated development environments*** (IDEs) provide richer software development platforms for Python, including one named IDLE that is included with the standard Python distribution. IDLE provides an embedded text-editor with support for displaying and editing Python code, and a basic debugger, allowing step-by-step execution of a program while examining key variable values.

＊许多集成开发环境（IDE）为Python提供了更加丰富的软件开发平台，其中包括标准Python发行版中集成的一个名为IDLE的IDE。IDLE提供了一个嵌入式的文本编辑器，支持显示和编辑Python代码，除此之外还支持单步调试。

1.1.2 Preview of a Python Program

＊Python程序预览

As a simple introduction, Code Fragment 1.1 presents a Python program that computes the grade-point average (GPA) for a student based on letter grades that are entered by a user. Many of the techniques demonstrated in this example will be discussed in the remainder of this chapter. At this point, we draw attention to a few high-level issues, for readers who are new to Python as a programming language.

＊Code Fragment 1.1提供了一个Python程序，它能够根据用户输入的字母等级计算学生的平均几点（GPA）。本示例中展示的许多内容将在本章的其余部分进行讨论。在这一点上，Python的新手需要注意几个高级的问题。

Python’s syntax relies heavily on the use of whitespace. Individual statements are typically concluded with a newline character, although a command can extend to another line, either with a concluding backslash character (\), or if an opening delimiter has not yet been closed, such as the { character in deﬁning value\_map.

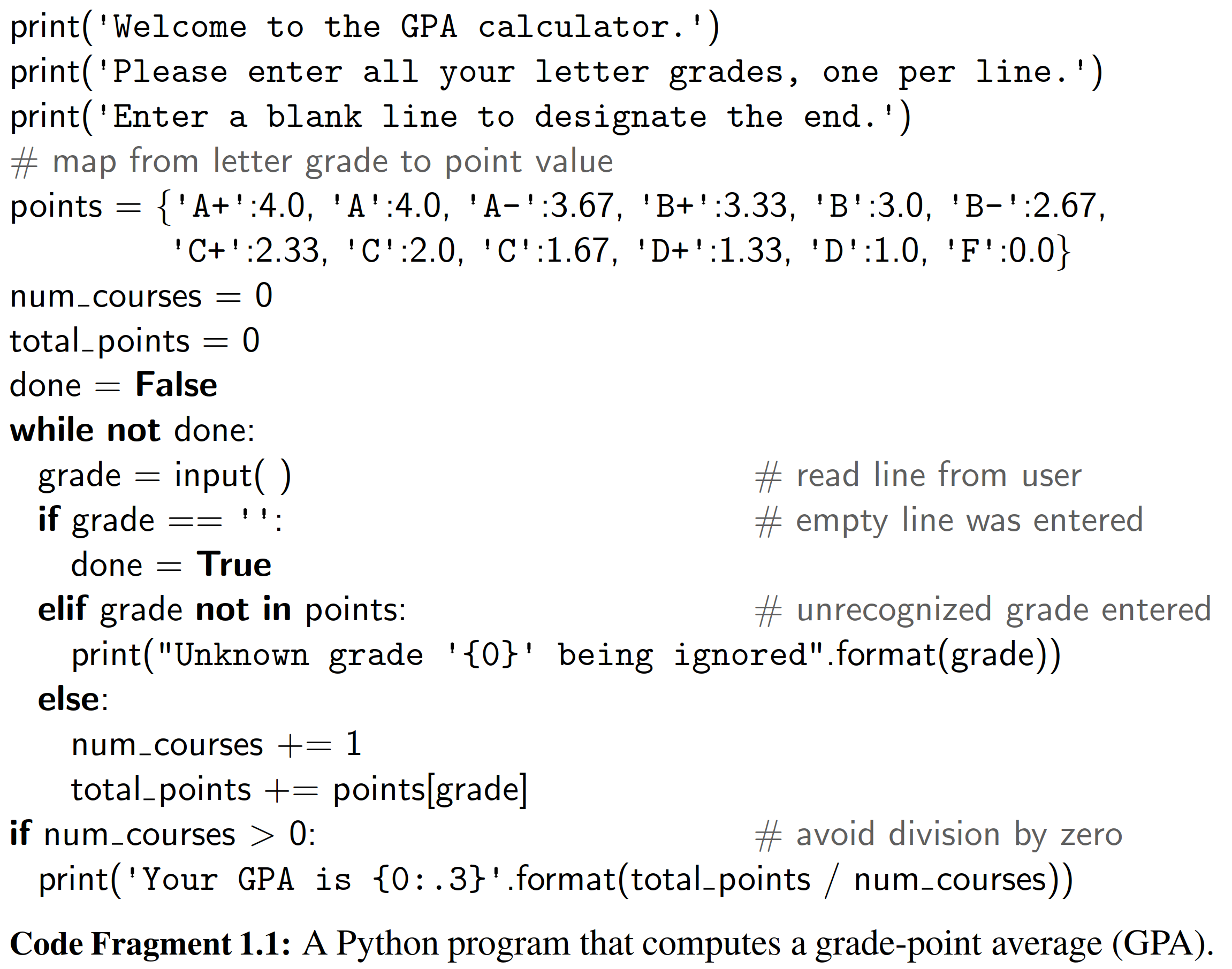
＊Python的语法严格依赖空格缩进。单个语句通常以换行符结尾，如果命令太长，那么可以通过反斜杠符号将语句扩展到另一行，例如定义value\_map的时候。

Whitespace is also key in delimiting the bodies of control structures in Python. Speciﬁcally, a block of code is indented to designate it as the body of a control structure, and nested control structures use increasing amounts of indentation. In Code Fragment 1.1, the body of the while loop consists of the subsequent 8 lines, including a nested conditional structure.

＊与此同时，空格也是Python中控制结构体的关键。特别地，一个代码块被缩进以将其指定为程序块的主体，而嵌套进其中的结构要在这一缩进级别上再次进行缩进，以此类推。在代码片段1.1中，while循环的主体由后面的8行组成，包括嵌套的条件结构。

Comments are annotations provided for human readers, yet ignored by the Python interpreter. The primary syntax for comments in Python is based on use of the # character, which designates the remainder of the line as a comment.

＊为读者提供的代码注释将会Python解释器忽略。Python中的注释是基于#字符的使用，它将该行的#之后的部分指定为注释。



1.2 Object in Python

＊1.2节 Python中的对象

Python is an object-oriented language and ***classes*** form the basis for all data types. In this section, we describe key aspects of Python’s object model, and we introduce Python’s built-in classes, such as the int class for integers, the ﬂoat class for ﬂoating-point values, and the str class for character strings. A more thorough presentation of object-orientation is the focus of Chapter 2.

＊Python是面向对象的语言，类是所有数据类型的基础。在本节中，我们将介绍Python对象模型的关键方面，并介绍Python的内置类，例如整数的int类，浮点值的float类和字符串的str类。第2章将更全面地介绍面向对象。

1.2.1 Identifiers, Objects, and the Assignment Statement

＊1.2.1节 标识符，对象与赋值语句

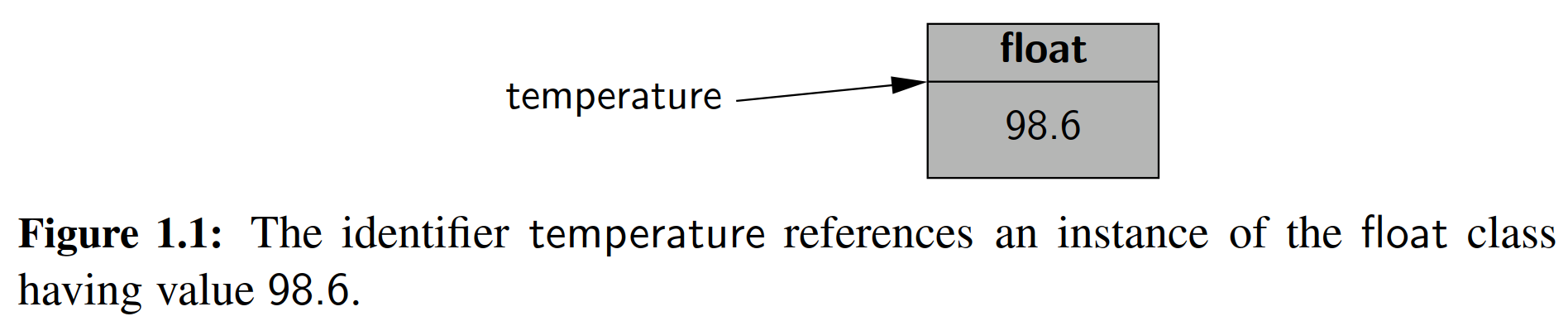
The most important of all Python commands is an ***assignment statement***, such as

＊最重要的Python命令是赋值命令，比如：

temperature = 98.6

This command establishes temperature as an ***identiﬁer*** (also known as a ***name***), and then associates it with the ***object*** expressed on the right-hand side of the equal sign, in this case a ﬂoating-point object with value 98.6. We portray the outcome of this assignment in Figure 1.1.

＊该命令将temperature设置为标识符（也称为名字），然后将其与等号右侧表示的对象相关联，在本例中，右侧的对象是值为98.6的浮点对象。我们在图1.1中描绘了这个语句的结果。

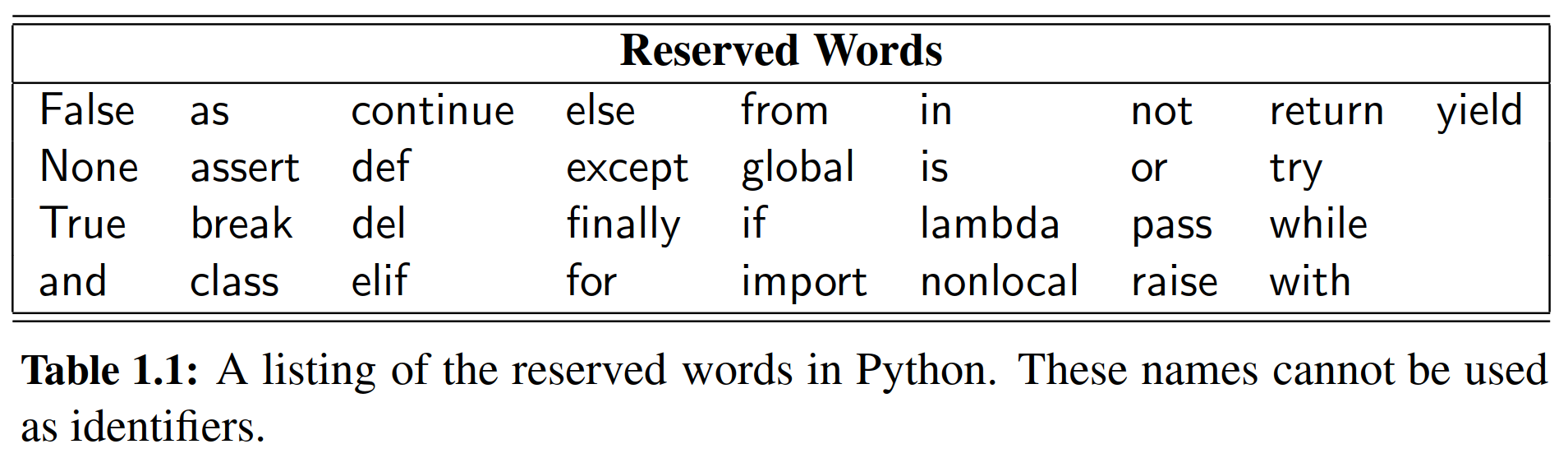


**Identiﬁers**

＊标识符

Identiﬁers in Python are ***case-sensitive***, so temperature and Temperature are distinct names. Identiﬁers can be composed of almost any combination of letters, numerals, and underscore characters (or more general Unicode characters). The primary restrictions are that an identiﬁer cannot begin with a numeral (thus 9lives is an illegal name), and that there are 33 specially reserved words that cannot be used as identiﬁers, as shown in Table 1.1.

＊Python中的标识符区分大小写，因此temperature和Temperature是不同的名称。标识符可以由字母，数字和下划线字符（或更一般的Unicode字符）的任意组合组成。主要限制是标识符不能以数字开头（因此，9lives是非法名称），并且有33个特别保留的单词不能用作标识符，如表1.1所示。



For readers familiar with other programming languages, the semantics of a Python identifier is most similar to a reference variable in Java or a pointer variable in C++. Each identifier is implicitly associated with the memory address of the object to which it refers. A Python identifier may be assigned to a special object named None, serving a similar purpose to a null reference in Java or C++.

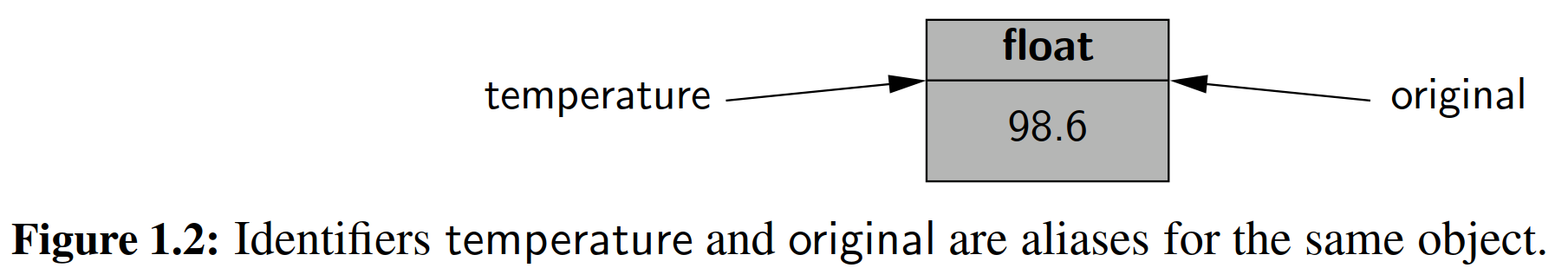
＊对于熟悉其他编程语言的读者而言，Python的标识符与Java语言中的引用以及C++语言中的指针变量很相似。每一个标识符都标记了它所指向的对象在内存中的地址。一个Python的标识符被赋值为一个特殊的对象——None，它的作用与Java和C++中的null相同。

Unlike Java and C++, Python is a ***dynamically typed*** language, as there is no advance declaration associating an identifier with a particular data type. An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type. Although an identifier has no declared type, the object to which it refers has a definite type. In our first example, the characters 98.6 are recognized as a floating-point literal, and thus the identifier temperature is associated with an instance of the float class having that value.

＊Python不像Java或者C++那样，它是一门动态的语言，因为它前期并没有把给定的标识符与某种固定的数据结构绑定。一个标识符可以和任意类型的对象进行绑定，并且之后，这个标识符可以和其他同类型或者不同类型的对象进行重新绑定。尽管一个标识符没有声明自己的类型，但是它指向的对象却有类型。在我们的第一个例子里，字符98.6在字面上被理解为一个浮点型，而temperature这个标识符就与拥有着这个数值的一个float类的实例联系起来了

A programmer can establish an ***alias*** by assigning a second identifier to an existing object. Continuing with our earlier example, Figure 1.2 portrays the result of a subsequent assignment, original = temperature.

＊通过标识符之间赋值的这种方式，一个程序员可以给一个已经存在的对象建立一个别名。继续沿用我们之前的例子，图1.2描述了后置赋值的结果，original = temperature.（言外之意就是，两个标识符都指向了内存中同一块区域。）



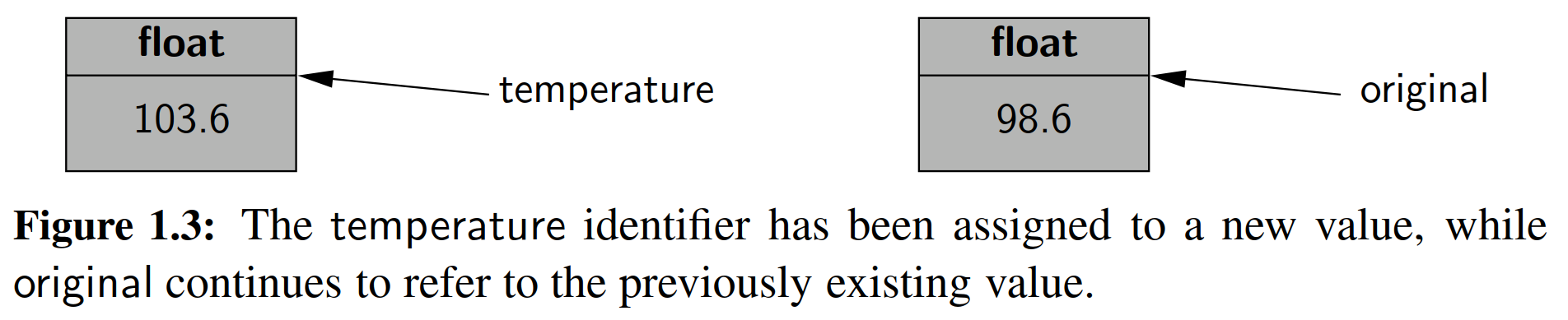
Once an alias has been established, either name can be used to access the underlying object. If that object supports behaviors that affect its state, changes enacted through one alias will be apparent when using the other alias (because they refer to the same object). However, if one of the *names* is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias. Continuing with our concrete example, we consider the command:

＊当一个对象的别名建立之后，原来的名字与别名都可以访问这个对象。如果对象方法中包含能够影响他自身状态的那种，那么通过引用一个别名而产生的变化自然也会作用在另外的别名上，因为它们指向的都是同一个对象。然而，如果这些别名中的一个被后续赋值语句赋予了新的值，那么这就不会影响其他的别名对象，而是中断了这个别名机制（而变成了一个新的对象）。还是举我们之前的实例，思考一下这个命令：

temperature = temperature + 5.0

The execution of this command begins with the evaluation of the expression on the right-hand side of the = operator. That expression, temperature + 5.0, is evaluated based on the existing binding of the name temperature, and so the result has value 103.6, that is, 98.6 + 5.0. That result is stored as a new floating-point instance, and only then is the name on the left-hand side of the assignment statement, temperature, (re)assignment to the result. The subsequent configuration is diagrammed in Figure 1.3. Of particular note, this last command had no effect on the value of the existing float instance that identifier original continues to reference.

＊这个语句的执行是从等号的右边开始的。temperature + 5.0这个语句基于temperature所既有的数值，所以运算结果是103.6，也就是98.6 + 5.0. 这个结果被储存在了一个新的浮点型实例中，而这之后才是等号左边元素的赋值，即temperature被赋予了一个新的浮点型对象。图1.3给出了后续的结构。值得注意的是，第二种命令对original所指向的浮点型实例并不起作用。



1.2.2 Creating and Using Objects

**Instantiation**

＊实例化

The process of creating a new instance of a class is known as ***instantiation***. In general, the syntax for instantiating an object is to invoke the ***constructor*** of a class. For example, if there were a class named Widget, we could create an instance of that class using a syntax such as w = Widget(), assuming that the constructor does not require any parameters. If the constructor does require parameters, we might use a syntax such as Widget(a, b, c) to construct a new instance.

＊创建类的新实例的过程称为实例化。通常，实例化对象的语法是调用类的构造函数。例如，如果有一个名为Widget的类，而且类的构造不需要参数，那我们就可以使用w = Widget()这个语句来创建该类的实例。如果构造函数确实需要参数，我们就要使用w = Widget(a,b,c)这样的语法构造一个类的实例。

Many of Python’s built-in classes (discussed in Section 1.2.3) support what is known as a ***literal*** form for designating new instances. For example, the command temperature = 98.6 results in the creation of a new instance of the ﬂoat class; the term 98.6 in that expression is a literal form. We discuss further cases of Python literals in the coming section.

＊许多Python的内置类（在1.2.3节中讨论）支持字面量形式。例如，使用语句temperature = 98.6可以创建一个浮点类的实例；该表达式中的第98.6是一个字面的形式。我们在下一节讨论更多有关Python字面量的案例。

From a programmer’s perspective, yet another way to indirectly create a new instance of a class is to call a function that creates and returns such an instance. For example, Python has a built-in function named sorted (see Section 1.5.2) that takes a sequence of comparable elements as a parameter and returns a new instance of the list class containing those elements in sorted order.

＊从程序员的角度来看，间接创建类的新实例的另一种方法是调用一个函数,该函数可以创建并返回一个实例。例如，Python里有一个名为sorted()的内置函数（参见第1.5.2节），它的参数是一个序列，而且要求这个序列中的元素是可以比较大小的。调用这个函数，会生成一个列表，表中元素是经过排序的。

**Calling Methods**

＊调用方法

Python supports traditional functions (see Section 1.5) that are invoked with a syntax such as sorted(data), in which case data is a parameter sent to the function. Python’s classes may also deﬁne one or more ***methods*** (also known as ***member functions***), which are invoked on a speciﬁc instance of a class using the dot (“.”) operator. For example, Python’s list class has a method named sort that can be invoked with a syntax such as data.sort(). This particular method rearranges the contents of the list so that they are sorted.

＊Python支持传统样式的函数调用，如sorted(data)（参见第1.5节），在这种情况下，data是参数。在Python的类里面也可以定义一个或多个方法（也称为成员函数），这些方法使用点（“.”）运算符在类的特定实例上发挥作用。例如，Python的列表类有一个名为sort的方法，可以使用data.sort()这样的语法进行类方法调用。这个sort方法将列表的内容进行重新排列，使之变得有序。

The expression to the left of the dot identiﬁes the object upon which the method is invoked. Often, this will be an identiﬁer (e.g., data), but we can use the dot operator to invoke a method upon the immediate result of some other operation. For example, if response identiﬁes a string instance (we will discuss strings later in this section), the syntax response.lower().startswith('y') ﬁrst evaluates the method call, response.lower(), which itself returns a new string instance, and then the startswith('y') method is called on that intermediate string.

＊点的左边是对象。而这个对象通常是一个标识符（例如data），我们可以使用点运算符调用类中方法从而对实例本身直接产生改变。例如，如果response是一个字符串实例（我们将在本节稍后讨论字符串），则response.lower().startswith('y')首先计算response.lower()，该方法调用本身返回一个新的字符串实例，然后在该过渡字符串上调用startswith('y')方法。

When using a method of a class, it is important to understand its behavior. Some methods return information about the state of an object, but do not change that state. These are known as ***accessors***. Other methods, such as the sort method of the list class, do change the state of an object. These methods are known as ***mutators*** or ***update methods***.

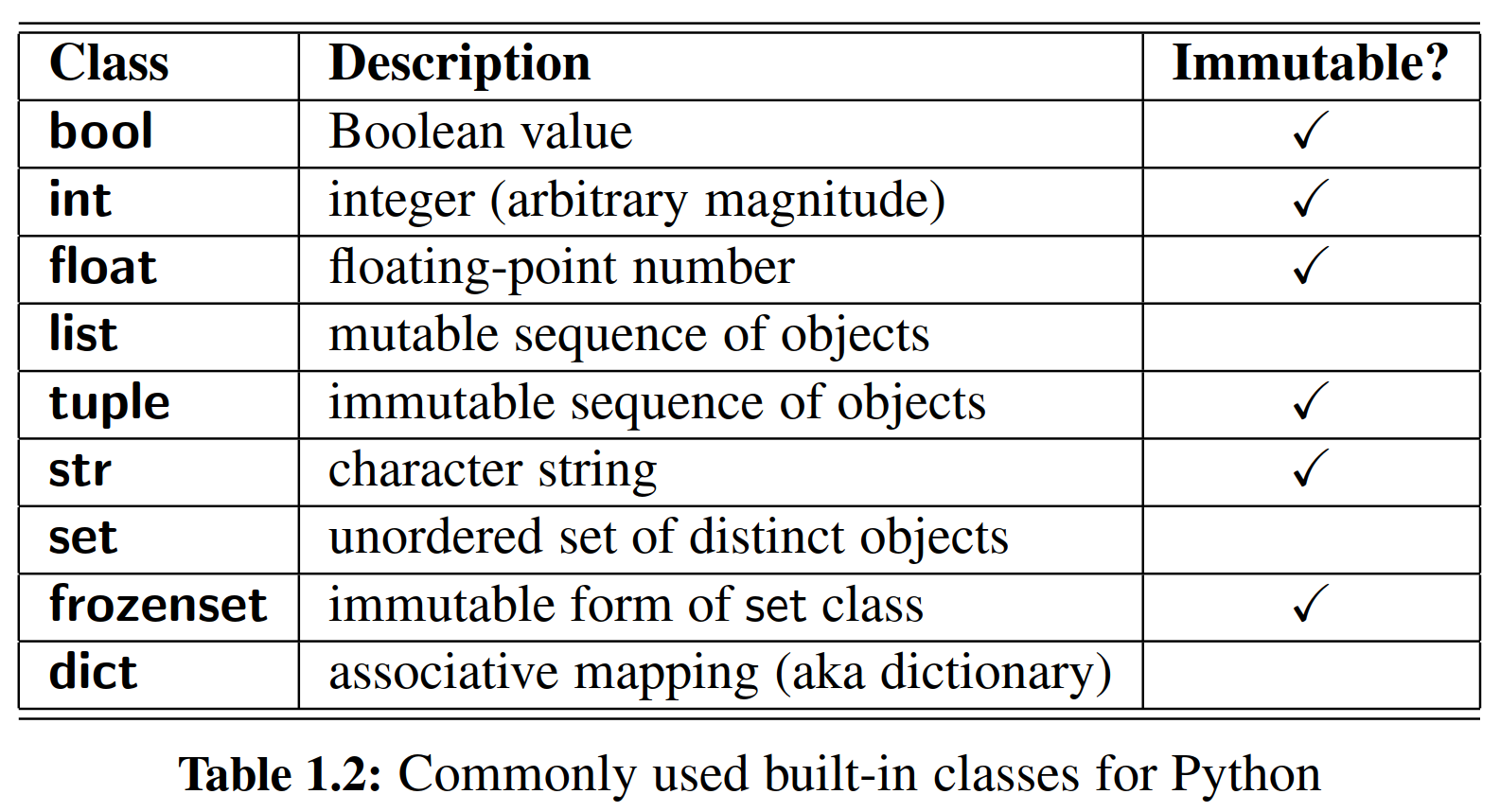
＊当使用类的方法时，了解方法的具体行为很重要。某些方法会返回对象的状态信息，但并不会更改该状态。这种方法被称为访问器。而其他方法，如列表类的排序方法，会改变对象的状态。这些方法称为变更器或更新方法。

1.2.3 Python’s Built-In Classes

＊Python的内建类

Table 1.2 provides a summary of commonly used, built-in classes in Python; we take particular note of which classes are mutable and which are immutable. A class is ***immutable*** if each object of that class has a ﬁxed value upon instantiation that cannot subsequently be changed. For example, the ﬂoat class is immutable. Once an instance has been created, its value cannot be changed (although an identiﬁer referencing that object can be reassigned to a different value).

＊表1.2提供了有关Python中常用的内置类的总结；我们要特别注意哪些类是可变的，哪些是不可变的。如果该类的每个对象在实例化后都具有固定值，而且不能随后更改，那么这个类是不可变的。例如，float类是不可变的。实例一经创建，其值不能被更改（尽管可以将指向该对象的标识符重新分配给不同的值）。



In this section, we provide an introduction to these classes, discussing their purpose and presenting several means for creating instances of the classes. Literal forms (such as 98.6) exist for most of the built-in classes, and all of the classes support a traditional constructor form that creates instances that are based upon one or more existing values. Operators supported by these classes are described in Section 1.3. More detailed information about these classes can be found in later chapters as follows: lists and tuples (Chapter 5); strings (Chapters 5 and 13, and Appendix A); sets and dictionaries (Chapter 10).

＊在本节中，我们将介绍这些类，讨论它们的存在意义以及介绍几种创建类实例的方法。大多数内置类都有字面形式（如98.6），同时所有类都支持像构建函数那样的构建形式，来创建基于一个或多个现有值的实例。这些类支持的运算符在第1.3节中有描述。有关这些类的更多详细信息，请参见以下章节：列表和元组（第5章）；字符串（第5章和第13章以及附录A）；集合和字典（第10章）。

**The bool Class**

＊布尔类

The bool class is used to manipulate logical (Boolean) values, and the only two instances of that class are expressed as the literals True and False. The default constructor, bool(), returns False, but there is no reason to use that syntax rather than the more direct literal form. Python allows the creation of a Boolean value from a nonboolean type using the syntax bool(foo) for value foo. The interpretation depends upon the type of the parameter. Numbers evaluate to False if zero, and True if nonzero. Sequences and other container types, such as strings and lists, evaluate to False if empty and True if nonempty. An important application of this interpretation is the use of a nonboolean value as a condition in a control structure.

＊bool类用于操纵逻辑（布尔）值，该类的唯一两个实例表示为True和False。构造函数bool()默认返回False，一般直接使用文字形式，而不是调用bool()函数。Python允许使用bool(foo)语句，将非布尔类的值，转化为一个布尔值。函数返回值取决于参数的类型：如果零，则评估为False，如果非零，则返回True。序列以及其他的容器类型，如字符串和列表，如果为空，则评估为False，如果非空，则评估为True。这种解释，使得非布尔值也可以作为控制结构中的判断条件。

**The int Class**

＊整数类

The **int** and **ﬂoat** classes are the primary numeric types in Python. The **int** class is designed to represent integer values with arbitrary magnitude. Unlike Java and C++, which support different integral types with different precisions (e.g., int, short, long), Python automatically chooses the internal representation for an integer based upon the magnitude of its value. Typical literals for integers include 0,137, and -23. In some contexts, it is convenient to express an integral value using binary, octal, or hexadecimal. That can be done by using a preﬁx of the number 0 and then a character to describe the base. Example of such literals are respectively 0b1011, 0o52, and 0x7f.

＊int和float类是Python中的基本数字类型。int类被设计为表示任意大小的整数值。不像Java和C++那样支持不同精度（例如，int，short，long）的整数类型，Python会根据其值的大小自动选择整数的占用空间。整数的典型样式包括0,137和-23。在某些情况下，使用二进制，八进制或十六进制表达整数值会更加方便。这种非十进制的表示方法可以这样写，用0作为前缀，其后跟着表示进位制的英文首字母小写，之后跟着数字。例子中的表达式分别为0b1011，0o52以及0x7f。

The integer constructor, int(), returns value 0 by default. But this constructor can be used to construct an integer value based upon an existing value of another type. For example, if f represents a ﬂoating-point value, the syntax int(f) produces the *truncated* value of f. For example, both int(3.14) and int(3.99) produce the value 3, while int(-3.9) produces the value -3. The constructor can also be used to parse a string that is presumed to represent an integral value (such as one entered by a user). If s represents a string, then int(s) produces the integral value that string represents. For example, the expression int('137') produces the integer value 137. If an invalid string is given as a parameter, as in int('hello'), a ValueError is raised (see Section 1.7 for discussion of Python’s exceptions). By default, the string must use base 10. If conversion from a different base is desired, that base can be indicated as a second, optional, parameter. For example, the expression int('7f ',16) evaluates to the integer 127.

＊整数构造函数int()默认返回值0。但是，此构造函数可用于根据另一类型的现有值构造整数值。例如，如果f表示浮点值，那么命令int(f)会返回f的整数部分截断值。例如，int(3.14)和int(3.99)都产生值3，而int(-3.9)则产生值-3。int()函数也可用于解释表示整数值的字符串（例如由用户输入的字符串）。如果s表示一个字符串，则int(s)将产生字符串表示的整数值。例如，表达式int('137')产生整数值137.如果将无效字符串作为参数给出，如int('hello')中，则会引发ValueError（有关Python的异常的讨论，请参见第1.7节）。默认情况下，字符串所表达的整数必须使用十进制。如果需要从不同的进位制进行转换，则该进位制应当被指示为第二个可选参数。例如，表达式int('7f',16)的计算结果为127。

**The float Class**

＊浮点类

The **ﬂoat** class is the sole ﬂoating-point type in Python, using a ﬁxed-precision representation. Its precision is more akin to a double in Java or C++, rather than those languages’ ﬂoat type. We have already discussed a typical literal form, 98.6. We note that the ﬂoating-point equivalent of an integral number can be expressed directly as 2.0. Technically, the trailing zero is optional, so some programmers might use the expression 2. to designate this ﬂoating-point literal. One other form of literal for ﬂoating-point values uses scientiﬁc notation. For example, the literal 6.022e23 represents the mathematical value 6.022 × 1023.

＊float类是Python中唯一的采用固定精度的浮点类型。它的精度更像Java或C++中的double类型。我们已经讨论了一个典型的字面形式，98.6。我们注意到，与整数相等的浮点数可以用2.0这种样式来表示。从计算机内部原理上讲，后缀0是可去的，所以一些程序员可能使用表达式2.来指定这个浮点数字。另外一种形式的浮点样式借鉴了科学计数法。例如，6.022e23表示数学值6.022×1023。

The constructor form of ﬂoat() returns 0.0. When given a parameter, the constructor attempts to return the equivalent ﬂoating-point value. For example, the call ﬂoat(2) returns the ﬂoating-point value 2.0. If the parameter to the constructor is a string, as with ﬂoat('3.14'), it attempts to parse that string as a ﬂoating-point value, raising a ValueError as an exception.

＊函数float()默认返回0.0。当给定一个参数时，构造函数尝试返回与之等值的浮点值。例如，调用float(2)返回浮点型2.0。如果函数float()的参数是一个字符串，比如float('3.14')，这时候函数会尝试将该字符串解析为浮点值，如果不匹配就会抛出ValueError异常。

**Sequence Types: The list, tuple, and str Classes**

＊序列类：列表类、元组类、字符类

The **list**, **tuple**, and **str** classes are ***sequence*** types in Python, representing a collection of values in which the order is signiﬁcant. The list class is the most general, representing a sequence of arbitrary objects (akin to an “array” in other languages). The tuple class is an ***immutable*** version of the list class, beneﬁting from a streamlined internal representation. The str class is specially designed for representing an immutable sequence of text characters. We note that Python does not have a separate class for characters; they are just strings with length one.

＊列表，元组和字符串类是Python中的序列类型，代表一个或多个有序的集合。列表类是最通用的，表示任意对象的序列（类似于其他语言中的“数组”）。元组类是列表类的不可变版本，因为内部表示很简单，所以有不少好处。字符串类专门用于表示不可变的文本字符序列。我们注意到，Python中没有单字符类；事实上单个字符只是长度为1的特殊字符串。

**The list Class**

＊列表类

A **list** instance stores a sequence of objects. A list is a ***referential*** structure, as it technically stores a sequence of references to its elements (see Figure 1.4). Elements of a list may be arbitrary objects (including the None object). Lists are ***array-based*** sequences and are ***zero-indexed***, thus a list of length n has elements indexed from 0 to n − 1 inclusive. Lists are perhaps the most used container type in Python and they will be extremely central to our study of data structures and algorithms. They have many valuable behaviors, including the ability to dynamically expand and contract their capacities as needed. In this chapter, we will discuss only the most basic properties of lists. We revisit the inner working of all of Python’s sequence types as the focus of Chapter 5.

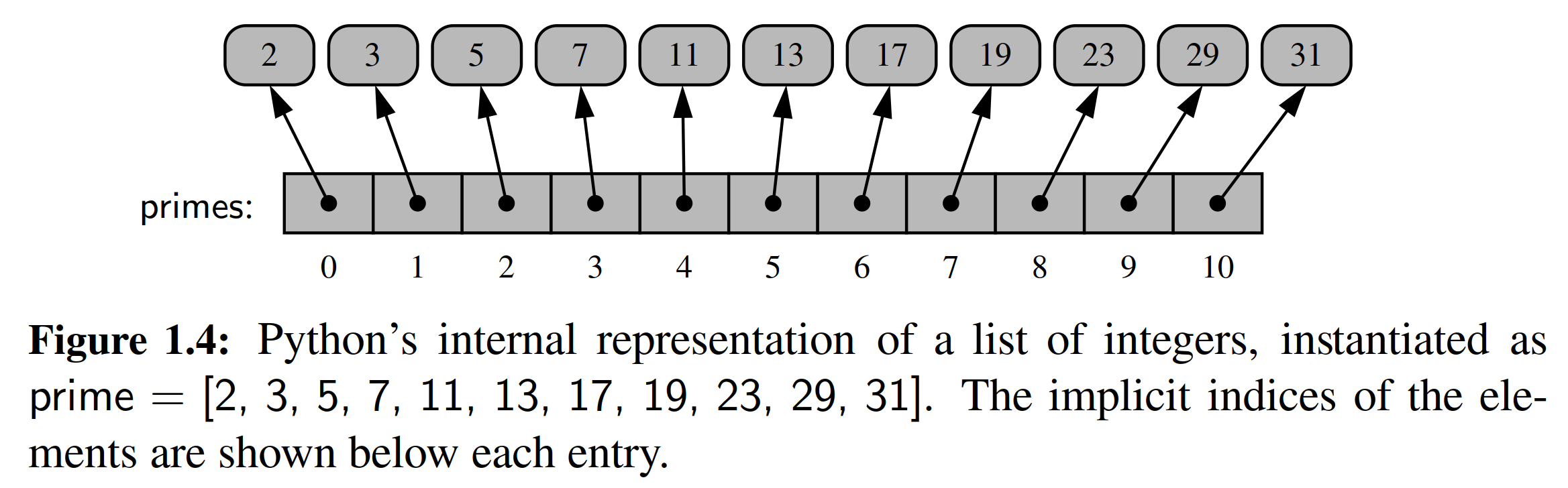
＊列表可以储存一个对象的实例序列。列表储存的是对象的地址（参见图1.4）。列表的元素可以是任意对象（包括None对象）。列表是基于数组的序列，并且是以0为下标开头的，因此长度为n的列表具有下标从0到n-1的元素。列表可能是Python中最常用的容器类型，它对我们对数据结构和算法的学习至关重要。列表有许多有价值的行为，包括动态变化。在本章中，我们将仅讨论列表的最基本属性。在第5章，我们将要重新审视Python所有序列类型的内部机制，而这也将是第五章的重点。

Python uses the characters [] as delimiters for a list literal, with[] itself being an empty list. As another example, ['red', 'green', 'blue'] is a list containing three string instances. The contents of a list literal need not be expressed as literals; if identiﬁers a and b have been established, then syntax [a,b] is legitimate.

＊Python使用字符[]作为列表文字的分隔符，而[]本身是一个空列表。另举一个例子，['red','green','blue']是一个包含三个字符串实例的列表。列表中内容的字面值不需要字面化表示；如果标识符a和b已经建立，则语句[a,b]就是合法的。

The list() constructor produces an empty list by default. However, the constructor will accept any parameter that is of an ***iterable*** type. We will discuss iteration further in Section 1.8, but examples of iterable types include all of the standard container types (e.g., strings, list, tuples, sets, dictionaries). For example, the syntax list('hello') produces a list of individual characters, ['h','e','l','l','o']. Because an existing list is itself iterable, the syntax backup = list(data) can be used to construct a new list instance referencing the same contents as the original.

＊函数list()默认生成一个空列表。但是，构造函数将接受任何可迭代类型的参数。我们将在1.8节中进一步讨论迭代，可迭代类型包括所有标准容器类型（例如，字符串，列表，元组，集合，字典）。例如，命令list('hello')可以生成单个字符['h'，'e'，'l'，'l'，'o']的列表。因为现有列表本身是可迭代的，所以语法backup = list(data)可用于新建一个与原始内容相同的新列表实例。

****

**The tuple Class**

＊元组类

The **tuple** class provides an immutable version of a sequence, and therefore its instances have an internal representation that may be more streamlined than that of a list. While Python uses the [] characters to delimit a list, parentheses delimit a tuple, with () being an empty tuple. There is one important subtlety. To express a tuple of length one as a literal, a comma must be placed after the element, but within the parentheses. For example, (17,) is a one-element tuple. The reason for this requirement is that, without the trailing comma, the expression (17) is viewed as a simple parenthesized numeric expression.

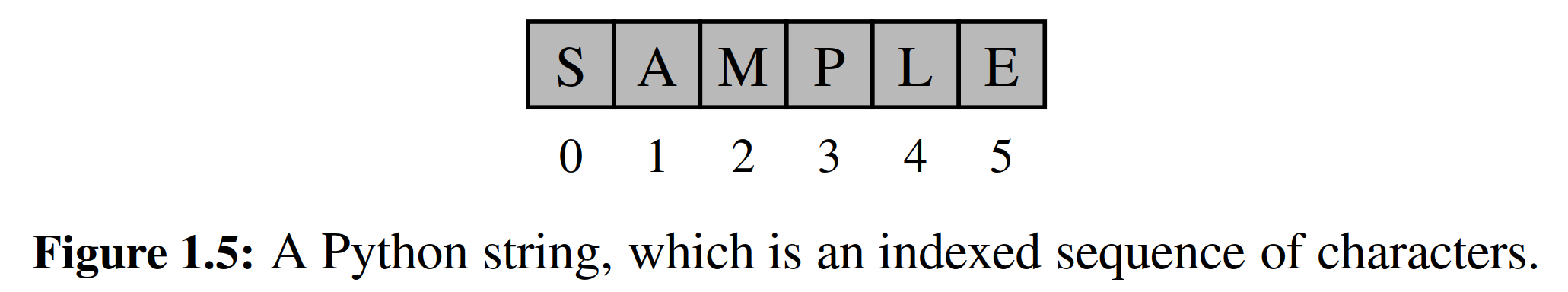
＊元组提供了一个不可变的序列版本，因此其表示或许比列表更简洁。Python使用[]字符来定义列表，()中定义元组，而()是一个空的元组。有一个重要的细节。要表达一个元素个数为1而且元素是一个整数的元组，必须把逗号放在元素后面、右括号的前面。例如，(17,)是一个单元素元组。之所以这么做，是因为(17)会被视为一个简单的括号数字表达式，而不是一个元组。

**The str class**

＊字符串类

Python’s str class is speciﬁcally designed to efﬁciently represent an immutable sequence of characters, based upon the Unicode international character set. Strings have a more compact internal representation than the referential lists and tuples, as portrayed in Figure 1.5.

＊Python的字符串类被专门设计为有效表示Unicode字符集的不可变字符序列。字符串具有比参考列表和元组更紧凑的内部表示，如图1.5所示。



String literals can be enclosed in single quotes, as in 'hello', or double quotes, as in "hello". This choice is convenient, especially when using another of the quotation characters as an actual character in the sequence, as in "Don't worry". Alternatively, the quote delimiter can be designated using a backslash as a so-called ***escape character***, as in 'Don\'t worry'. Because the backslash has this purpose, the backslash must itself be escaped to occur as a natural character of the string literal, as in C:\\Python\\ , for a string that would be displayed as C:\Python\. Other commonly escaped characters are \n for newline and \t for tab. Unicode characters can be included, such as '20\u20AC' for the string 20€.

＊字符串文字可以包含在单引号中，如'hello',当然双引号也可以，如"hello"。后一个选择是合适的，特别是当使用另一个引号作为序列中的实际字符时，比如"Don't worry"。或者采用另一种方法，使用反斜杠作为转义字符来指定引号分隔符，如'Don\'t worry'。因为反斜杠具有这种功能，所以反斜杠作为实际字符时就要采取双写策略，比如'C:\\Python\\'，会显示为C:\Python\。其他通常转义的字符有用作换行符的\n，用作制表符的\t。甚至转移字符机制里面还包括了Unicode，例如'20\u20AC'会被解释成20€。

Python also supports using the delimiter ''' or """ to begin and end a string literal. The advantage of such triple-quoted strings is that newline characters can be embedded naturally (rather than escaped as \n). This can greatly improve the readability of long, multiline strings in source code. For example, at the beginning of Code Fragment 1.1, rather than use separate print statements for each line of introductory output, we can use a single print statement, as follows:

＊在Python中，还可以使用分隔符'''或"""来开始和结束一个字符串文字。这样优点是可以直接在编辑器里enter换行（而不是转义为\n）。这可以大大提高源代码里那种行数多、容量大的字符串的可读性，例如，在代码片段1.1的开头，不必为每一行的内容写一个打印语句，我们使用一个打印语句就可以解决问题，如下所示：

print("""Welcome to the GPA calculator.

Please enter all your letter grades, one per line.

Enter a blank line to designate the end.""")

**The set and frozenset Classes**

＊集合列与冻结集合类

Python’s **set** class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a speciﬁc element is contained in the set. This is based on a data structure known as a ***hash table*** (which will be the primary topic of Chapter 10). However, there are two important restrictions due to the algorithmic underpinnings. The ﬁrst is that the set does not maintain the elements in any particular order. The second is that only instances of ***immutable*** types can be added to a Python set. Therefore, objects such as integers, ﬂoating-point numbers, and character strings are eligible to be elements of a set. It is possible to maintain a set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The frozenset class is an immutable form of the set type, so it is legal to have a set of frozensets.

＊Python的集合类表示的是数学概念上的集合，即元素的集合，不重复，无顺序。使用集合而不用列表主要优点是它具有一种高度优化的方法来检查集合中是否包含特定元素。这种优化的方法基于哈希表这一数据结构（这将是第10章的主要内容）。然而，由于算法的约束，集合类的使用存在两个重要的限制。首先是集合不以任何特定顺序保存元素。第二个是只有不可变类型的实例才可以添加到集合类中。诸如整数，浮点数和字符串的对象可以成为集合的元素。可以把好多元组放到集合里，但是集合本身或者列表是不能作为集合的元素的，因为列表和集合是可变的。而冻结集合是一种不可变集合类型，所以允许把冻结集合作为一个集合的元素。

Python uses curly braces {and} as delimiters for a set, for example, as {17} or {'red','green','blue'}. The exception to this rule is that {} does not represent an empty set; for historical reasons, it represents an empty dictionary (see next paragraph). Instead, the constructor syntax set() produces an empty set. If an iterable parameter is sent to the constructor, then the set of distinct elements is produced. For example, set('hello') produces {'h','e','l','o'}.

＊Python使用花括号{and}作为一个集合的分隔符，例如{17}或{'red'，'green'，'blue'}。然而{}不表示空集；它代表其实是一个空字典（见下一段），而这是由历史原因造成的。相反，函数set()产生一个空集。如果将一个可迭代对象作为参数发送给函数set()，那么会生成一个集合。例如，执行命令set('hello')会生成{'h'，'e'，'l'，'o'}。

**The dict Classes**

＊字典类

Python’s dict class represents a dictionary, or mapping, from a set of distinct keys to associated values. For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student’s name, address, and course grades). Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.

＊Python的字典类表示一个从一组不同的键值到另外一组关联值的映射。例如，字典可以从唯一的学生证号码映射到学生记录（例如学生的姓名，地址和课程等级）。Python使用与集合几乎相同的方法实现了字典类，只是增加了关联值的存储。

A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form {} produces an empty dictionary. A nonempty dictionary is expressed using a comma-separated series of key: value pairs. For example, the dictionary {'ga': 'Irish', 'de': 'German'} maps 'ga' to 'Irish' and 'de' to 'German'.

＊标识字典也用花括号，但是因为字典在集合之前被引入，所以{}代表一个空字典。非空字典用逗号分隔一键值对，键值对之间用‘:’表示映射关系。例如，{'ga': 'Irish', 'de': 'German'}将'ga'映射到'Irish'，将'de'到'German'。

The constructor for the dict class accepts an existing mapping as a parameter, in which case it creates a new dictionary with identical associations as the existing one. Alternatively, the constructor accepts a sequence of key-value pairs as a parameter, as in dict(pairs) with pairs = [('ga','Irish'), ('de', 'German')].

＊字典类的构造函数接受现有映射作为参数，在这种情况下，它将创建一个与现有映射相同的新字典。同样地，构造函数接受一个键值对序列作为参数，如在dict(pairs)中使用pairs= [('ga','Irish'),('de','German')]。

1.3 Expressions, Operations, and Precedence

＊表达式、运算符以及优先级

In the previous section, we demonstrated how names can be used to identify existing objects, and how literals and constructors can be used to create instances of built-in classes. Existing values can be combined into larger syntactic ***expressions*** using a variety of special symbols and keywords known as ***operators***. The semantics of an operator depends upon the type of its operands. For example, when a and b are numbers, the syntax a + b indicates addition, while if a and b are strings, the operator indicates concatenation. In this section, we describe Python’s operators in various contexts of the built-in types.

＊在上一节中，我们演示了如何用标识符来区分对象，以及如何使用语句和构造函数来创建内置类的实例。已有的值可以使用各种运算符组合成较大的合成表达式。相同运算符的实际含义取决于其可操作的类型。例如，当a和b是数字时，语法a + b表示数的加法，而如果a和b是字符串，则运算符表示将两个字符串合并。在本节中，我们将介绍有关Python内建类的多种运算符，以及这些运算符在不同的上下文里的意义。

We continue, in Section 1.3.1, by discussing ***compound expressions***, such as a + b \* c, which rely on the evaluation of two or more operations. The order in which the operations of a compound expression are evaluated can affect the overall value of the expression. For this reason, Python defines a specific order of precedence for evaluating operators, and it allows a programmer to override this order by using explicit parentheses to group subexpressions.

＊在1.3.1节中，我们继续讨论依赖于两个或多个操作的联合表达式，如a + b \* c之类的。复合表达式的运算顺序会影响表达式的总体值。因此，Python定义了用于评估运算符的特定优先顺序，并允许程序员通过使用括号对子表达式进行编排而覆盖此顺序。

**Logical Operators**

＊逻辑运算符

Python supports the following keyword operators for Boolean values:

＊对于布尔类型的值，Python提供了一下几种操作运算符：

|  |  |  |
| --- | --- | --- |
| **not** | | unary negation  ＊否定，单变量操作 |
| **and** | conditional and  ＊蕴含式与 |
| **or** | | conditional or  ＊蕴含式或 |

The **and** and **or** operators ***short-circuit***, in that they do not evaluate the second operand if the result can be determined based on the value of the ﬁrst operand. This feature is useful when constructing Boolean expressions in which we ﬁrst test that a certain condition holds (such as a reference not being **None**), and then test a condition that could have otherwise generated an error condition had the prior test not succeeded.

＊对于and和or操作如果结果可以基于第一个变量而得到，则它们就不会计算第二个变量。这个特性在构造布尔表达式的时候很有用，比如我们首先测试一个特定条件（例如指向是不是为空），然后测试一个可能会在先前的测试不成功的情况下产生错误条件的条件。

**Equality Operators**

＊相等判断符

Python supports the following operators to test two notions of equality:

＊Python为两个变量的是否等值提供了以下运算符：

|  |  |
| --- | --- |
| **is** | same identity  ＊指向相同 |
| **is not** | different identity  ＊指向不同 |
| **==** | equivalent  ＊相等 |
| **!=** | not equivalent  ＊不等 |

The expression a **is** b evaluates to True, precisely when identiﬁers a and b are aliases for the same object. The expression a == b tests a more general notion of equivalence. If identiﬁers a and b refer to the same object, then a == b should also evaluate to True. Yet a == b also evaluates to True when the identiﬁers refer to different objects that happen to have values that are deemed equivalent. The precise notion of equivalence depends on the data type. For example, two strings are considered equivalent if they match character for character. Two sets are quivalent if they have the same contents, irrespective of order. In most programming situations, the equivalence tests == and != are the appropriate operators; use of is and is not should be reserved for situations in which it is necessary to detect true aliasing.

＊当标识符a和b指向同一对象时，表达式a is b的值为True，。a == b表达了更一般的等价概念。如果标识符a和b指向相同的对象，则a == b的计算结果也为True。当a，b指向具有相同值的不同对象时，a == b也会评估为True。等价与否取决于数据类型。例如，如果两个字符串相同，则这两个字符串被认为是等价的。如果两个集合具有相同的内容，则二者是等价的，而且不论元素顺序如何。在大多数编程情况下，==和!=是相反的运算符；使用is和is not应该被用在检测别名设置的情形下。

**Comparison Operators**

＊比较运算符

Data types may deﬁne a natural order via the following operators:

＊通过以下的运算符可以确定数据的大小顺序。

|  |  |
| --- | --- |
| < | less than  ＊小于 |
| <= | less than or equal to  ＊小于等于 |
| > | greater than  ＊大于 |
| >= | greater than or equal to  ＊大于等于 |

These operators have expected behavior for numeric types, and are deﬁned lexicographically, and case-sensitively, for strings. An exception is raised if operands have incomparable types, as with 5 < 'hello'.

＊这些操作符可以比较数字的大小，也可以对字符串进行了字典排序和大小写排序。如果变量的类型不相同，比如5 < 'hello'，那么就会引发异常。

**Arithmetic Operators**

＊算数运算符

Python supports the following arithmetic operators:

＊Python支持如下的算数运算符

|  |  |
| --- | --- |
| + | addition  ＊加 |
| － | subtraction  ＊减 |
| \* | multiplication  ＊乘 |
| / | true division  ＊除 |
| // | integer division  ＊整除 |
| % | the modulo operator  ＊取余、取模 |

The use of addition, subtraction, and multiplication is straightforward, noting that if both operands have type int, then the result is an int as well; if one or both operands have type ﬂoat, the result will be a ﬂoat.

＊加法，减法和乘法是简单的，如果两个操作数都是整型，那么结果也是整型；如果变量中的一个或两个是浮点型，则结果是浮点型。

Python takes more care in its treatment of division. We ﬁrst consider the case in which both operands have type int, for example, the quantity 27 divided by 4. In mathematical notation, . In Python, the / operator designates ***true division***, returning the ﬂoating-point result of the computation. Thus, 27 / 4 results in the ﬂoat value 6.75. Python supports the pair of operators // and % to perform the integral calculations, with expression 27 // 4 evaluating to int value 6 (the mathematical ﬂoor of the quotient), and expression 27 % 4 evaluating to int value 3, the remainder of the integer division. We note that languages such as C, C++, and Java do not support the // operator; instead, the / operator returns the truncated quotient when both operands have integral type, and the result of true division when at least one operand has a ﬂoating-point type.

＊Python在除法方面更加谨慎。我们首先考虑两个操作数都是整型的情况，例如27除以4。在数学中，27 ÷ 4 = 6 3/4 = 6.75。在Python中，/运算符指的是真正的除法，返回浮点型结果。因此，27/4的浮点型表示是6.75。对于整型，Python支持//和％操作，27 // 4的计算值为6，这是商的向下取整；27 ％ 4的结果为3，这是取模运算。我们注意到C，C++和Java等语言不支持//操作符；不过在这些语言中，当两个变量都是整型时，/运算符返回取整的商，当存在一个变量是浮点类型时，返回真除法的结果。

Python carefully extends the semantics of // and % to cases where one or both operands are negative. For the sake of notation, let us assume that variables n and m represent respectively the ***dividend*** and ***divisor*** of a quotient n/m, and that q = n // m and r = n % m. Python guarantees that q \* m + r will equal n. We already saw an example of this identity with positive operands, as 6 ∗ 4 + 3 = 27. When the divisor m is positive, Python further guarantees that 0 ≤ r < m. As a consequence, we ﬁnd that −27 // 4 evaluates to −7 and −27 % 4 evaluates to 1, as (−7) ∗ 4 + 1 = −27. When the divisor is negative, Python guarantees that m < r ≤ 0. As an example, 27 // −4 is −7 and 27 % −4 is −1, satisfying the identity 27 = (−7) ∗ (−4) + (−1).

＊Python将//和％扩展到一个或者两个变量为负数的情况。为了避免引入新的符号，我们假定变量n和m分别表示被除数和除数，q = n // m和r = n％m。Python保证 q \* m + r = n。我们已经看到了正数的例子，如6 \* 4 + 3 = 27。当除数m为正时，Python进一步保证0 ≤ r < m。因此，我们发现-27 // 4 的结果为-7，-27 ％ 4的结果为1，如(-7) \* 4 + 1 = -27。当除数为负时，Python保证m < r ≤0。例如，27 // -4是-7，27％-4是-1，满足27 = (-7) \* (-4)+（-1）。

The conventions for the // and % operators are even extended to ﬂoating- point operands, with the expression q = n // m being the integral ﬂoor of the quotient, and r = n % m being the “remainder” to ensure that q \* m + r equals n. For example, 8.2 // 3.14 evaluates to 2.0 and 8.2 % 3.14 evaluates to 1.92, as 2.0 ∗ 3.14 + 1.92 = 8.2.

＊//和％运算符甚至可以扩展到浮点数，q = n // m是商的向下取整，r = n％m是“余数”，以确保qm + r等于n。例如，8.2 // 3.14等于2.0，8.2％3.14等于1.92，满足2.0 \* 3.14 + 1.92 = 8.2。

**Bitwise Operators**

＊位操作符

Python provides the following bitwise operators for integers:

＊Python对整型数字提供了以下几种位操作方式：

|  |  |
| --- | --- |
| ∼ | bitwise complement (preﬁx unary operator)  ＊ |
| & | bitwise and  ＊按位与 |
| | | bitwise or  ＊按位或 |
| ˆ | bitwise exclusive-or  ＊按位异或 |
| << | shift bits left, ﬁlling in with zeros  ＊左移，0填充空位 |
| >> | shift bits right, ﬁlling in with sign bit  ＊右移，符号填充即算数右移 |

**Sequence Operators**

＊序列运算符

Each of Python’s built-in sequence types (str, tuple, and list) support the following operator syntaxes:

＊每个Python的内置序列类型（str，tuple和list）都支持以下运算符：

|  |  |
| --- | --- |
| s[j] | element at index *j*  ＊处于下标j位置的元素 |
| s[start:stop] | slice including indices [start, stop)  ＊从start到stop（但是不包含stop）的序列片段 |
| s[start: stop: step] | slice including indices start, start + step, start + 2\*step, ..., up to but not equaling or stop  ＊步长为step（第三个参数）的片段 |
| s + t | concatenation of sequences  ＊序列连接 |
| k \* s | shorthand for s + s + s + ... (k times)  ＊ |
| val **in** s | containment check  ＊包含检测 |
| val **not** **in** s | non-containment check  ＊不包含检测 |

Python relies on ***zero-indexing*** of sequences, thus a sequence of length n has elements indexed from 0 to n − 1 inclusive. Python also supports the use of ***negative indices***, which denote a distance from the end of the sequence; index −1 denotes the last element, index −2 the second to last, and so on. Python uses a ***slicing*** notation to describe subsequences of a sequence. Slices are described as half-open intervals, with a start index that is included, and a stop index that is excluded. For example, the syntax data[3:8] denotes a subsequence including the ﬁve indices: 3, 4, 5, 6, 7. An optional “step” value, possibly negative, can be indicated as a third parameter of the slice. If a start index or stop index is omitted in the slicing notation, it is presumed to designate the respective extreme of the original sequence.

＊Python依赖于序列的零索引，因此长度为n的序列具有从0到n-1的下标索引。Python还支持使用负数下标，借此倒着数的顺序；下标-1表示最后一个元素，索引-2表示最后一个元素，依此类推。Python使用切片符号来描述序列的子序列。片段被划分为半开间隔，其中包含起始索引，但是不包含终结的元素。例如，data[3:8]表示包含五个索引的子序列：3，4，5，6，7。可选的step值可被指示为第三个参数。如果在分片符号中省略了开始索引或停止索引，则假设它们指定原始序列的相应首尾。

Because lists are mutable, the syntax s[j] = val can be used to replace an element at a given index. Lists also support a syntax, del s[j], that removes the designated element from the list. Slice notation can also be used to replace or delete a sublist.

＊因为列表是可变的，语法s[j] = val可以用于替换给定索引处的元素。列表还支持从列表中删除指定元素的语法del s[j]。切片符号也可用于替换或删除子列表。

The notation val in s can be used for any of the sequences to see if there is an element equivalent to val in the sequence. For strings, this syntax can be used to check for a single character or for a larger substring, as with amp in example.

＊s中的符号val可用于任何序列，以查看序列中是否存在等价于val的元素。对于字符串，此语法可用于检查单个字符或子串，与示例中的放大器一样。

All sequences deﬁne comparison operations based on lexicographic order, performing an element by element comparison until the ﬁrst difference is found. For example, [5, 6, 9] < [5, 7] because of the entries at index 1. Therefore, the following operations are supported by sequence types:

＊所有序列定义了基于字典顺序的比较操作，通过元素比较执行元素，直到找到第一个不同元素。例如，由于下标1中的条目，[5,6,9] < [5,7]。因此，序列类型支持以下操作：

|  |  |
| --- | --- |
| s == t | equivalent (element by element)  ＊字符串相等 |
| s != t | not equivalent  ＊字符串不相等 |
| s < t | lexicographically less than  ＊字典排序下的小于 |
| s <= t | lexicographically less than or equal to  ＊字典排序下的小于等于 |
| s > t | lexicographically greater than  ＊字典排序下的大于 |
| s >= t | lexicographically greater than or equal to  ＊字典排序下的大于等于 |

**Operators for Sets and Dictionaries**

＊对于集合与字典的操作符

Sets and frozensets support the following operators:

＊集合与冻结集合都支持下面的操作符

|  |  |
| --- | --- |
| key **in** s | containment check  ＊包含检测 |
| key **not** **in** s | non-containment check  ＊不包含检测 |
| s1 == s2 | s1 is equivalent to s2  ＊集合相等 |
| s1 != s2 | s1 is not equivalent to s2  ＊集合不等 |
| s1 <= s2 | s1 is subset of s2  ＊s1是s2的子集 |
| s1 < s2 | s1 is proper subset of s2  ＊s1是s2的真子集 |
| s1 >= s2 | s1 is superset of s2  ＊s1是s2的父集 |
| s1 > s2 | s1 is proper superset of s2  ＊＊s1是s2的真父集 |
| s1 | s2 | the union of s1 and s2  ＊集合的并 |
| s1 & s2 | the intersection of s1 and s2  ＊集合的交 |
| s1 − s2 | the set of elements in s1 but not s2  ＊集合的差 |
| s1 ˆ s2 | the set of elements in precisely one of s1 or s2  ＊集合的对称差（剔除交集） |

Note well that sets do not guarantee a particular order of their elements, so the comparison operators, such as <, are not lexicographic; rather, they are based on the mathematical notion of a subset. As a result, the comparison operators deﬁne a partial order, but not a total order, as disjoint sets are neither “less than,” “equal to,” or “greater than” each other. Sets also support many fundamental behaviors through named methods (e.g., add, remove); we will explore their functionality more fully in Chapter 10.

＊请注意，集合不能保证其元素的特定顺序，因此比较运算符（例如<）的结果不是依据字典排序得出的，而是基于子集的概念而得出的。比较运算符定义了一个偏序关系，而不是一个总顺序，因为不相交集之间，既不是“小于”，“等于”或“大于”。集合还通过命名方法支持许多基本行为（例如，add，remove）；我们将在第10章中更全面地探讨其功能。

Dictionaries, like sets, do not maintain a well-deﬁned order on their elements. Furthermore, the concept of a subset is not typically meaningful for dictionaries, so the dict class does not support operators such as <. Dictionaries support the notion of equivalence, with d1 == d2 if the two dictionaries contain the same set of key- value pairs. The most widely used behavior of dictionaries is accessing a value associated with a particular key k with the indexing syntax, d[k]. The supported operators are as follows:

＊词典，也像集合一样，不会在其元素上保持一个明确的秩序。此外，子集的概念对于字典通常不是有意义的，因此dict类不支持诸如<的比较操作符。字典支持等价的概念，如果两个词典包含相同的键值对集合，则使用d1 == d2来表示二者相等。字典中使用最广泛的行为是使用索引语法d[k]访问与特定密钥k相关联的值。支持的操作符如下：

|  |  |
| --- | --- |
| d[key] | value associated with given key  ＊与key值相关联的value |
| d[key] = value | set (or reset) the value associated with given key  ＊将value链接到key上面 |
| **del** d[key] | remove key and its associated value from dictionary  ＊将k, v pair (key, d[key])删除 |
| key **in** d | containment check  ＊包含检查 |
| key **not** **in** d | non-containment check  ＊不包含检查 |
| d1 == d2 | d1 is equivalent to d2  ＊字典等价 |
| d1 != d2 | d1 is not equivalent to d2  ＊字典不等价 |

Dictionaries also support many useful behaviors through named methods, which we explore more fully in Chapter 10.

＊词典还支持许多有用的行为，我们在第10章中更全面地探讨。

**Extended Assignment Operators**

＊扩展的赋值运算符

Python supports an extended assignment operator for most binary operators, for example, allowing a syntax such as count += 5. By default, this is a shorthand for the more verbose count = count + 5. For an immutable type, such as a number or a string, one should not presume that this syntax changes the value of the existing object, but instead that it will reassign the identiﬁer to a newly constructed value. (See discussion of Figure 1.3.) However, it is possible for a type to redeﬁne such semantics to mutate the object, as the list class does for the += operator.

＊Python支持大多数的二元运算符的扩展赋值运算符，例如，count + = 5。默认情况下，这是count = count + 5的缩写。对于不可变类型，例如数字或字符串，不要认为这个语法改变了现有对象的值，事实上是将其重新分配给一个新构造的值。（见图1.3的讨论）当然，对于可变类这就是可行的了，就像列表类对于+=操作符一样。

alpha = [1,2,3]

beta = alpha

# an alias for alpha

beta += [4,5]

# extends the original list with two more elements

beta = beta + [6, 7]

# reassigns beta to a new list [1,2,3,4,5,6,7]

print(alpha)

# will be [1, 2, 3, 4, 5]

This example demonstrates the subtle difference between the list semantics for the syntax beta += foo versus beta = beta + foo.

＊此示例演示了语法beta += foo与beta = beta + foo的列表语义之间的微妙差异。

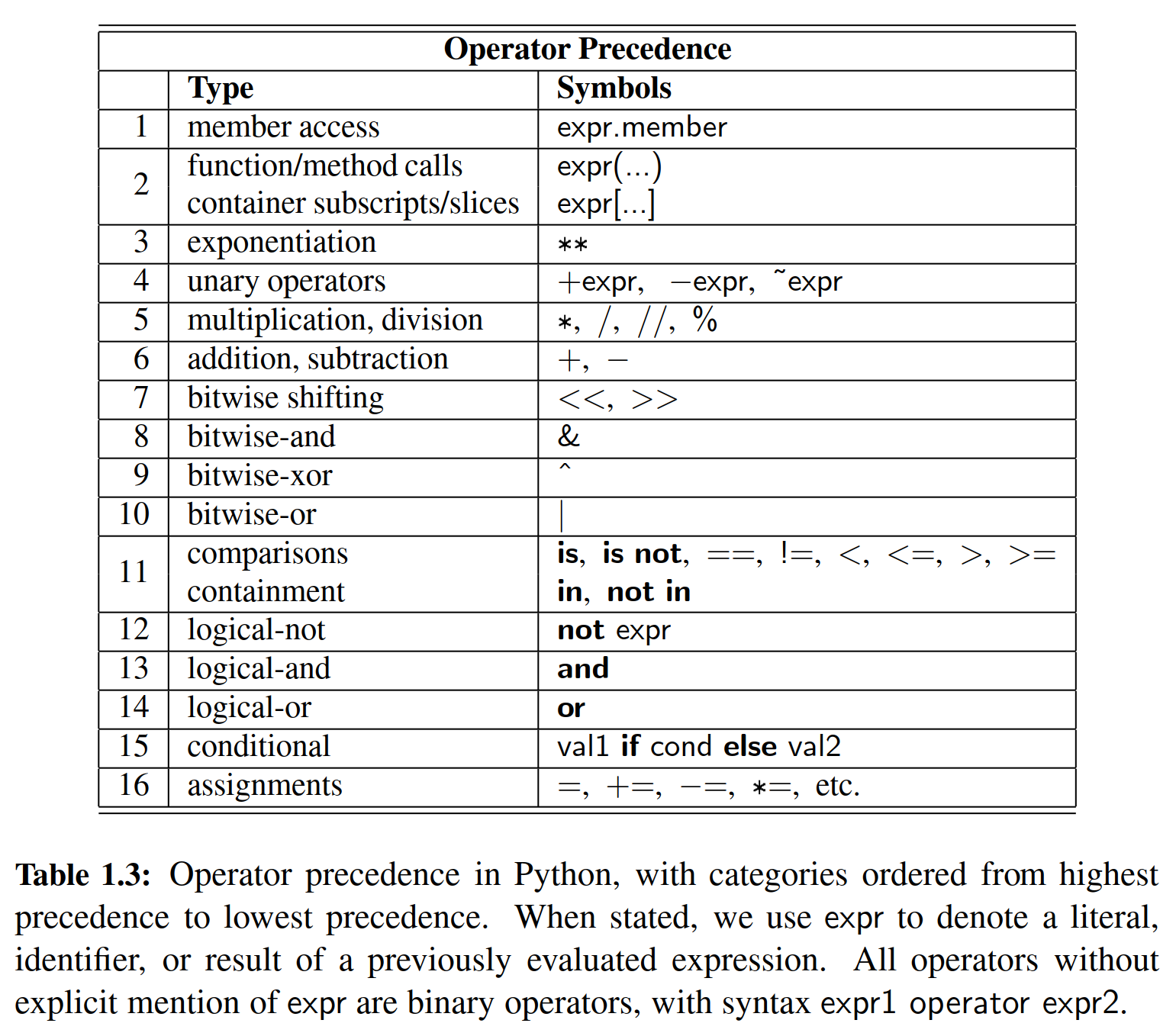
1.3.1 Compound Expressions and Operator Precedence

Programming languages must have clear rules for the order in which compound expressions, such as 5 + 2 \* 3, are evaluated. The formal order of precedence for operators in Python is given in Table 1.3. Operators in a category with higher precedence will be evaluated before those with lower precedence, unless the expression is otherwise parenthesized. Therefore, we see that Python gives precedence to multiplication over addition, and therefore evaluates the expression 5 + 2 \* 3 as 5 + (2 \* 3), with value 11, but the parenthesized expression (5 + 2) \* 3 evaluates to value 21. Operators within a category are typically evaluated from left to right, thus 5 − 2 + 3 has value 6. Exceptions to this rule include that unary operators and exponentiation are evaluated from right to left.

＊编程语言必须具有明确的规则，用于对哪些复合表达式（如5 + 2 \* 3）进行计算。Python中运算符的正式顺序是在表1.3中给出的。具有较高优先级的类别中的运算符将在优先级较低的类别之前进行计算，除非表达式被另外表示。因此，我们看到，Python给出了乘法加法的优先级，因此将5 + 2 \* 3的表达式计算为5 +（2 \* 3），值为11，但括号表达式（5 + 2）\* 3计算为值21。类别中的运算符通常从左到右进行计算，因此5 - 2 + 3等于6。但是这个规则也有例外，一元运算符和幂运算会从右到左进行计算。

Python allows a ***chained assignment***, such as x = y = 0, to assign multiple identiﬁers to the rightmost value. Python also allows the ***chaining*** of comparison operators. For example, the expression 1 <= x + y <= 10 is evaluated as the compound (1 <= x + y) **and** (x + y <= 10), but without computing the inter- mediate value x + y twice.

＊Python允许链式赋值，例如x = y = 0会将多个标识符分配给最右边的值。Python还允许比较运算符的链式计算。例如，表达式1 <= x + y <= 10被认为是(1 <= x + y) and (x + y <= 10)，但是后者会计算中间值x + y 两次。



1.4 Control Flow

＊

In this section, we review Python’s most fundamental control structures: conditional statements and loops. Common to all control structures is the syntax used in Python for deﬁning blocks of code. The colon character is used to delimit the beginning of a block of code that acts as a body for a control structure. If the body can be stated as a single executable statement, it can technically placed on the same line, to the right of the colon. However, a body is more typically typeset as an ***indented block*** starting on the line following the colon. Python relies on the indentation level to designate the extent of that block of code, or any nested blocks of code within. The same principles will be applied when designating the body of a function (see Section 1.5), and the body of a class (see Section 2.3).

＊在本节中，我们将回顾Python最基本的控制结构：条件语句和循环。所有控制结构的共同之处在于Python用于定义代码块的语法。冒号字符用于控件结构代码块的开头。如果主语句为单个可执行语句，则它可以被放置在冒号右侧的同一行上。然而，一个主体通常会在冒号之后的行上以缩进的块开始排版。Python依赖于缩进级别来指定代码块的范围，或任何嵌套的代码块。在指定功能体（参见第1.5节）和类（见第2.3节）时，将应用与之相同的原理。

1.4.1 Conditionals

＊条件语句

Conditional constructs (also known as **if** statements) provide a way to execute a chosen block of code based on the run-time evaluation of one or more Boolean expressions. In Python, the most general form of a conditional is written as follows:

＊条件结构（也称为if语句）提供了一种基于一个或多个布尔表达式计算结果而有选择地执行代码块的方法。在Python中，条件的最通用形式如下所示：

**if** *ﬁrst\_condition*:

*ﬁrst\_body*

**elif** second condition:

*second\_body*

**elif** *third\_condition*:

*third\_body*

**else**:

*fourth\_body*

Each condition is a Boolean expression, and each body contains one or more commands that are to be executed conditionally. If the ﬁrst condition succeeds, the ﬁrst body will be executed; no other conditions or bodies are evaluated in that case. If the ﬁrst condition fails, then the process continues in similar manner with the evaluation of the second condition. The execution of this overall construct will cause precisely one of the bodies to be executed. There may be any number of **elif** clauses (including zero), and the ﬁnal **else** clause is optional. As described on page 7, nonboolean types may be evaluated as Booleans with intuitive meanings. For example, if response is a string that was entered by a user, and we want to condition a behavior on this being a nonempty string, we may write

＊每个条件都是一个布尔表达式，每个主体都包含一个或多个按照条件执行的命令。如果第一个条件语句为真，第一个语句将被执行；在这种情况下，不再计算其他条件。如果第一个条件被判断为假，则该过程将会从第二个判断继续进行，以此类推。整体结构的执行将会导致正确的一个语句被执行。允许有任何数量的elif子句（包括零个），最后的else子句是可选的。如第7页所示，非布尔类型可以被评估为具有直观含义的布尔值。例如，如果判断值是由用户输入的字符串，并且我们想要将此行为设置为非空字符串，我们可能会将

**if** response:

as a shorthand for the equivalent,

＊作为以下语句等效的缩写

**if** response != ' ':

As a simple example, a robot controller might have the following logic:

＊简单举个例子，机器控制器可能具有以下逻辑：

**if** door\_is\_closed:

open\_door()

advance()

Notice that the ﬁnal command, advance(), is not indented and therefore not part of the conditional body. It will be executed unconditionally (although after opening a closed door).

＊请注意，最终命令advance()不缩进，因此不是条件体的一部分。它将无条件地执行（虽然打开一个关闭的门之后）。

We may nest one control structure within another, relying on indentation to make clear the extent of the various bodies. Revisiting our robot example, here is a more complex control that accounts for unlocking a closed door.

＊我们可以将一个控制结构嵌套在另一个控制结构中，而我们通过在已有缩进之上再次进行缩进，以便明确各个语句的范围。回顾我们的机器人示例，这里是一个更复杂的控制，它解释了如何打开上了锁的门。

**if** door\_is\_closed:

**if** door\_is\_locked:

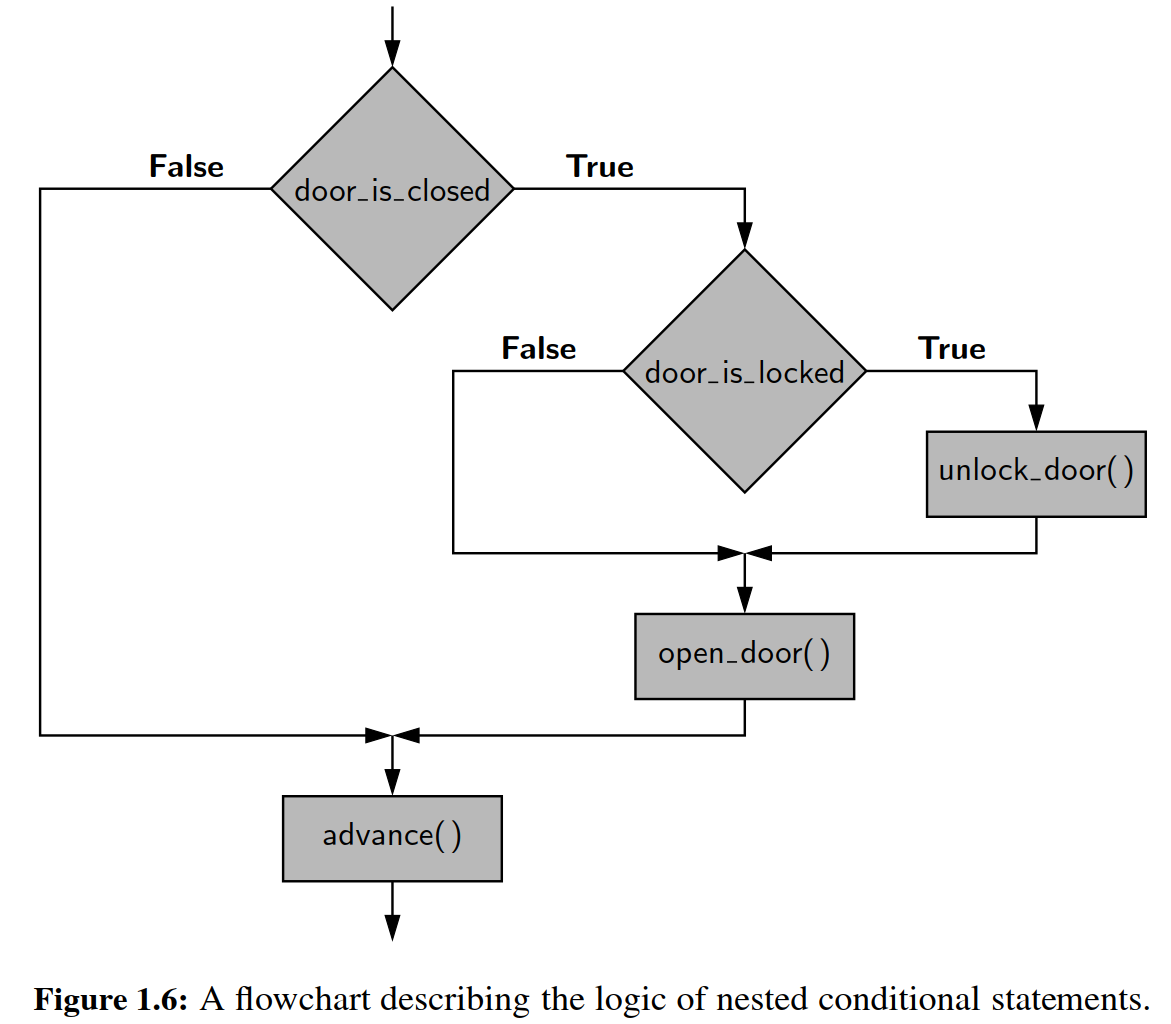
unlock\_door()

open\_door()

advance()

The logic expressed by this example can be diagrammed as a traditional ***ﬂowchart***, as portrayed in Figure 1.6.

＊该示例表示的逻辑可以用传统流图绘制出来，如图1.6所示。



1.4.2 Loops

＊循环

Python offers two distinct looping constructs. A **while** loop allows general repetition based upon the repeated testing of a Boolean condition. A **for** loop provides convenient iteration of values from a deﬁned series (such as characters of a string, elements of a list, or numbers within a given range). We discuss both forms in this section.

＊Python提供了两个不同的循环结构。while循环提供了基于布尔条件的重复测试的一般重复。for循环提供了一些列值的迭代循环（如字符串的所有字符，列表的所有元素或给定范围内的所有数字）。我们在本节讨论两种形式。

**While Loops**

＊while循环

The syntax for a **while** loop in Python is as follows:

＊while循环语句如下所示：

**while** *condition*:

*body*

As with an **if** statement, condition can be an arbitrary Boolean expression, and *body* can be an arbitrary block of code (including nested control structures). The execution of a while loop begins with a test of the Boolean condition. If that condition evaluates to True, the body of the loop is performed. After each execution of the body, the loop condition is retested, and if it evaluates to True, another iteration of the body is performed. When the conditional test evaluates to False (assuming it ever does), the loop is exited and the ﬂow of control continues just beyond the body of the loop.

＊与if语句一样，条件可以是任意的布尔表达式，而body可以是任意的代码块（包括嵌套的控制结构）。while循环的执行开始于布尔条件的测试。如果该条件为True，则执行循环的正文。在每次执行body后，重新测试循环条件，如果为True，则执行body的另一个迭代。当条件测试评估为False（假设它是）时，循环被退出，并且控制流程仅在循环体内截止。

As an example, here is a loop that advances an index through a sequence of characters until ﬁnding an entry with value X or reaching the end of the sequence.

＊举个例子，这里是一个循环，它通过一系列的字符来推进一个索引，直到找到一个具有值X条目，或者找不到而返回。

j = 0

**while** j < len(data) **and** data[j] != 'X':

j += 1

The len function, which we will introduce in Section 1.5.2, returns the length of a sequence such as a list or string. The correctness of this loop relies on the short- circuiting behavior of the and operator, as described on page 12. We intentionally test j < len(data) to ensure that j is a valid index, prior to accessing element data[j]. Had we written that compound condition with the opposite order, the evaluation of data[j] would eventually raise an IndexError when 'X' is not found. (See Section 1.7 for discussion of exceptions.)

＊我们将在1.5.2节介绍len函数，该函数会返回一个序列的长度，如列表或字符串的长度。此循环的正确性取决于第12页所述的运算符的短路行为。在访问元素data[j]之前，我们有意测试j < len(data)以确保j是有效索引。如果我们交换两个条件的顺序，则当没有找到“X”时，data[j]的计算结果将最终引发IndexError。（有关例外情况的讨论，请参见第1.7节。）

As written, when this loop terminates, variable j’s value will be the index of the leftmost occurrence of 'X', if found, or otherwise the length of the sequence (which is recognizable as an invalid index to indicate failure of the search). It is worth noting that this code behaves correctly, even in the special case when the list is empty, as the condition j < len(data) will initially fail and the body of the loop will never be executed.

＊如所写的，当该循环终止时，变量j的值将是找到的最开始的那个“X”的下标，否则为序列的长度（其被识别为指示搜索失败的无效索引）。值得注意的是，即使在列表为空的特殊情况下，该代码也是正确的正确，因为条件j < len(data)将首先被判断为错误，循环的主体将永远不会被执行。

**For Loops**

＊for 循环

Python’s **for**-loop syntax is a more convenient alternative to a while loop when iterating through a series of elements. The **for**-loop syntax can be used on any type of ***iterable*** structure, such as a list, tuple, str, set, dict, or ﬁle (we will discuss iterators more formally in Section 1.8). Its general syntax appears as follows.

＊Python的for循环语法是一个更方便的替代while循环的方法，尤其是当遍历一系列元素时，while循环。for循环语法可用于任何类型的可迭代结构，例如list，tuple，str，set，dict或者file（我们将在1.8节更正式地讨论迭代器）。其一般语法如下所示。

**for** element **in** iterable:

body # body may refer to 'element' as an identiﬁer

For readers familiar with Java, the semantics of Python’s for loop is similar to the “for each” loop style introduced in Java 1.5.

＊对于熟悉Java的读者，Python for循环的语义与Java 1.5中引入的“for each”循环类似。

As an instructive example of such a loop, we consider the task of computing the sum of a list of numbers. (Admittedly, Python has a built-in function, sum, for this purpose.) We perform the calculation with a for loop as follows, assuming that data identiﬁes the list:

＊作为这样一个循环的指导性例子，我们考虑计算数字列表的和的任务。（对于这个目的，Python有一个内置函数，sum）我们使用for循环执行以下计算，假设列表名为data：

total = 0

**for** val **in** data:

total += val # note use of the loop variable, val

The loop body executes once for each element of the data sequence, with the identiﬁer, val, from the for-loop syntax assigned at the beginning of each pass to a respective element. It is worth noting that val is treated as a standard identiﬁer. If the element of the original data happens to be mutable, the val identiﬁer can be used to invoke its methods. But a reassignment of identiﬁer val to a new value has no effect on the original data, nor on the next iteration of the loop.

＊循环体对于数据序列的每个元素执行一次，使用从每个通过开始时分配给相应元素的循环合成的标识符val。值得注意的是，val被视为一个标准的标识符。如果原始数据的元素恰巧是可变的，则可以使用val标识来调用其方法。但是将识别符重新分配到一个新的值对原始数据不会有任何影响，同样也不会影响循环的下一次迭代。

As a second classic example, we consider the task of ﬁnding the maximum value in a list of elements (again, admitting that Python’s built-in max function already provides this support). If we can assume that the list, data, has at least one element, we could implement this task as follows:

＊作为第二个典型例子，我们考虑在元素列表中确定最大值的任务（Python的内置max函数已经提供了这种支持）。如果我们可以假设data至少有一个元素，那么我们可以如下实现这个任务：

biggest = data[0] # as we assume nonempty list

**for** val **in** data:

if val > biggest:

biggest = val

Although we could accomplish both of the above tasks with a while loop, the for-loop syntax had an advantage of simplicity, as there is no need to manage an explicit index into the list nor to author a Boolean loop condition. Furthermore, we can use a for loop in cases for which a while loop does not apply, such as when iterating through a collection, such as a set, that does not support any direct form of indexing.

＊尽管我们可以用while循环来完成上述任务，但for循环语法具有简洁性上面的优势，因为不需要在列表中管理显式索引，也不需要创建一个布尔循环条件。此外，我们可以在不适用while循环的情况下使用for循环，例如当遍历不支持任何直接索引形式的集合（如set）时。

**Index-Based For Loops**

＊基于下标的for循环

The simplicity of a standard for loop over the elements of a list is wonderful; how- ever, one limitation of that form is that we do not know where an element resides within the sequence. In some applications, we need knowledge of the index of an element within the sequence. For example, suppose that we want to know *where* the maximum element in a list resides.

＊循环列表元素的简单性非常好；然而，该表单的一个限制是我们不知道元素在序列中的位置。在某些应用中，我们需要知道序列中元素的下标。比如我们想知道列表中最大值元素的位置。

Rather than directly looping over the elements of the list in that case, we prefer to loop over all possible indices of the list. For this purpose, Python provides a built-in class named range that generates integer sequences. (We will discuss generators in Section 1.8.) In simplest form, the syntax range(n) generates the series of n values from 0 to n - 1. Conveniently, these are precisely the series of valid indices into a sequence of length n. Therefore, a standard Python idiom for looping through the series of indices of a data sequence uses a syntax,

＊在这种情况下，我们不是直接循环列表中的元素，而是更好地循环列表中的所有可能的下标。为此，Python提供了一个内置的名为range的类，它生成整数序列。（我们将在1.8节中讨论生成器。）最简单的形式是，range(n)生成从0到n - 1的n个值。这些正好是长度为n的序列中的一系列有效索引。因此，用于循环一系列索引的标准Python语法如下：

**for** j **in** range(len(data)):

In this case, identiﬁer j is not an element of the data—it is an integer. But the expression data[j] can be used to retrieve the respective element. For example, we can ﬁnd the *index* of the maximum element of a list as follows:

＊在这种情况下，标识符j不是data的元素，它是一个整数。但表达式data[j]可用于检索相应的元素。例如，我们可以找到列表的最大元素的索引：

big index = 0

**for** j **in** range(len(data)):

**if** data[j] > data[big index]:

big index = j

**Break and Continue Statements**

＊break与continue语句

Python supports a break statement that immediately terminate a while or for loop when executed within its body. More formally, if applied within nested control structures, it causes the termination of the most immediately enclosing loop. As a typical example, here is code that determines whether a target value occurs in a data set:

＊Python支持break语句，break一旦执行，会立即终止一段while或for循环。更正式地，如果在嵌套控制结构中应用，则会导致该break与句所在的小循环的终止。作为一个典型的例子，这里是一个确定目标值是否在数据集中的代码：

found = False

**for** item **in** data:

**if** item == target:

found = **True**

break

Python also supports a **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

＊Python还支持continue语句，这个命令导致循环体的当前迭代停止，但循环的后续循环按预期进行。

We recommend that the break and continue statements be used sparingly. Yet, there are situations in which these commands can be effectively used to avoid introducing overly complex logical conditions.

＊我们建议小心谨慎地使用break和continue。然而，有些情况可以有效地使用这些命令来避免引入过于复杂的逻辑条件。

1.5 Functions

＊函数

In this section, we explore the creation of and use of functions in Python. As we did in Section 1.2.2, we draw a distinction between ***functions*** and ***methods***. We use the general term *function* to describe a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as sorted(data). We use the more speciﬁc term method to describe a member function that is invoked upon a speciﬁc object using an object-oriented message passing syntax, such as data.sort(). In this section, we only consider pure functions; methods will be explored with more general object-oriented principles in Chapter 2.

＊在本节中，我们将探讨Python中函数的创建和使用。正如我们在1.2.2节中所做的那样，我们对功能和方法进行了区分。我们使用通用术语来描述在没有特定类的或类实例（例如sorted(data)）的情况下调用的传统的无状态函数。我们使用更具体的术语来描述在面向对象方式中，对对象上调用成员函数时候的信息传递（如data.sort()）。在本节中，我们只考虑纯函数；第二章将探讨更为普遍的面向对象原则的方法。

We begin with an example to demonstrate the syntax for deﬁning functions in Python. The following function counts the number of occurrences of a given target value within any form of iterable data set.

＊我们从一个例子开始，演示Python中定义函数的语法。以下函数计算任何形式的可迭代数据集中给定目标值的出现次数。

**def** count(data, target):

n = 0

**for** item **in** data:

**if** item == target: # found a match

n += 1

**return** n

The ﬁrst line, beginning with the keyword **def**, serves as the function’s ***signature***. This establishes a new identiﬁer as the name of the function (count, in this example), and it establishes the number of parameters that it expects, as well as names identifying those parameters (data and target, in this example). Unlike Java and C++, Python is a dynamically typed language, and therefore a Python signature does not designate the types of those parameters, nor the type (if any) of a return value. Those expectations should be stated in the function’s documentation (see Section 2.2.3) and can be enforced within the body of the function, but misuse of a function will only be detected at run-time.

＊第一行以关键字def开头，用作函数的声明。这建立了一个新的标识符，它作为函数的名称（在本例中为count），并且它建立了它需要的参数，以及标识这些参数的名称（在本例中为data和target）。与Java和C++不同，Python是一种动态类型的语言，因此Python不指定这些参数的类型，也不指定返回值的类型（如果有）。这些参数、返回值的意义应该在函数文档中说明（见第2.2.3节），而且这些参数可以在函数体被调度，但只能在运行时才能被检测到功能的滥用。

The remainder of the function deﬁnition is known as the ***body*** of the function. As is the case with control structures in Python, the body of a function is typically expressed as an indented block of code. Each time a function is called, Python creates a dedicated ***activation record*** that stores information relevant to the current call. This activation record includes what is known as a ***namespace*** (see Section 1.10) to manage all identiﬁers that have ***local scope*** within the current call. The namespace includes the function’s parameters and any other identiﬁers that are deﬁned locally within the body of the function. An identiﬁer in the local scope of the function caller has no relation to any identiﬁer with the same name in the caller’s scope (although identiﬁers in different scopes may be aliases to the same object). In our ﬁrst example, the identiﬁer n has scope that is local to the function call, as does the identiﬁer item, which is established as the loop variable.

＊函数定义的其余部分称为函数体。与Python中的控制结构一样，函数的主体通常表示为缩进的代码块。每次调用函数时，Python会创建一个专用的记录，用于存储与当前调用相关的信息。此激活记录包括所谓的命名空间（参见第1.10节），以管理在当前调用中具有本地范围的所有标识符。命名空间包括函数的参数和在函数体内定义的任何其他标识符。函数调用者的标识符与函数中的同名标识符无关（尽管不同作用域中的标识符可能是同一对象的两个别名）。在我们的第一个例子中，n的作用范围仅仅在函数中起作用，同时被确定为循环变量。

**Return Statement**

＊

A **return** statement is used within the body of a function to indicate that the function should immediately cease execution, and that an expressed value should be returned to the caller. If a return statement is executed without an explicit argument, the None value is automatically returned. Likewise, None will be returned if the ﬂow of control ever reaches the end of a function body without having executed a return statement. Often, a return statement will be the ﬁnal command within the body of the function, as was the case in our earlier example of a count function. However, there can be multiple return statements in the same function, with conditional logic controlling which such command is executed, if any. As a further example, consider the following function that tests if a value exists in a sequence.

＊在函数体内使用return语句来表示该函数应该立即停止执行，并且一个表达的值应该返回给调用者。如果在没有显式参数的情况下执行return语句，返回None。同样，如果控制流已经到达函数体的末尾而没有执行返回语句，则返回None。通常，返回语句将是函数体内的最终命令，就像我们前面的count函数示例一样。但是，在同一个函数中可以有多个返回语句，如在条件控制中写多个返回语句（如果有的话）。作为另一个例子，以下函数用来测试序列中是否存在值。

**def** contains(data, target):

**for** item **in** target:

**if** item == target: # found a match

**return** True

**return** False

If the conditional within the loop body is ever satisﬁed, the return True statement is executed and the function immediately ends, with True designating that the target value was found. Conversely, if the for loop reaches its conclusion without ever ﬁnding the match, the ﬁnal return False statement will be executed.

＊如果循环体中的条件满足，则执行返回True语句，并且函数立即结束，True表明找到目标值。相反，如果for循环没有完成匹配，则将最终执行False语句。

1.5.1 Information Passing

＊信息传递

To be a successful programmer, one must have clear understanding of the mechanism in which a programming language passes information to and from a function. In the context of a function signature, the identiﬁers used to describe the expected parameters are known as ***formal parameters***, and the objects sent by the caller when invoking the function are the ***actual parameters***. Parameter passing in Python follows the semantics of the standard ***assignment statement***. When a function is invoked, each identiﬁer that serves as a formal parameter is assigned, in the function’s local scope, to the respective actual parameter that is provided by the caller of the function.

＊要成为一名成功的程序员，人们必须清楚地了解编程语言将信息传递到函数和从函数传递信息的机制。在函数的上下文中，用于描述预期参数的标识符称为形式参数，调用者在调用函数时发送的对象是实际参数。Python中的参数传递遵循标准赋值语句的语义。当函数被调用时，作为形参的每个标识符在函数的本地范围内被实参赋值。

For example, consider the following call to our count function from page 23:

＊例如，考虑对我们的count函数的以下调用：

prizes = count(grades, 'A' )

Just before the function body is executed, the actual parameters, grades and 'A', are implicitly assigned to the formal parameters, data and target, as follows:

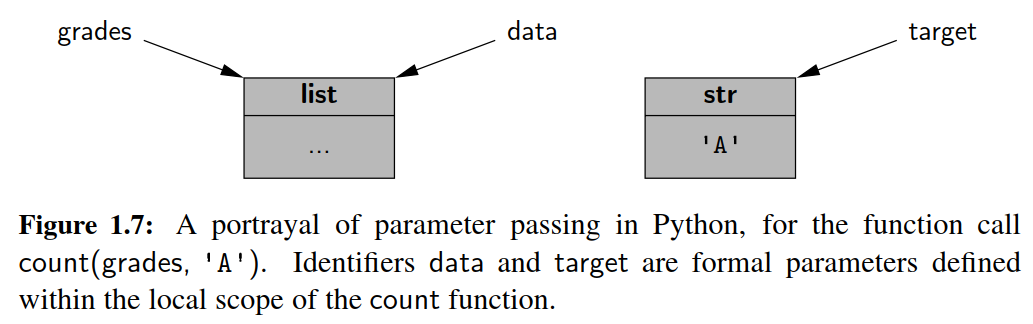
＊在函数体执行之前，系统将实参grades和“A”分配给形参data和targe，如下所示：

data = grades

target = 'A'

These assignment statements establish identiﬁer data as an alias for grades and target as a name for the string literal A. (See Figure 1.7.)

＊赋值语句将data设置为grades的别名，同理，将target其作为字符串A的名字。（见图1.7）



The communication of a return value from the function back to the caller is similarly implemented as an assignment. Therefore, with our sample invocation of prizes = count(grades, 'A' ), the identiﬁer prizes in the caller’s scope is assigned to the object that is identiﬁed as n in the return statement within our function body.

＊从函数返回到调用也是通过赋值来实现的。因此，通过我们的例子调用prizes = count(grade, 'A')，prizes标识符被赋值给函数体的返回值n。

An advantage to Python’s mechanism for passing information to and from a function is that objects are not copied. This ensures that the invocation of a function is efﬁcient, even in a case where a parameter or return value is a complex object.

＊Python向函数传递信息的一个优点是对象不被复制。这确保函数的调用是有效的，即使在参数或返回值是复杂对象的情况下也是如此。

**Mutable Parameters**

＊可变的参数

Python’s parameter passing model has additional implications when a parameter is a mutable object. Because the formal parameter is an alias for the actual parameter, the body of the function may interact with the object in ways that change its state. Considering again our sample invocation of the count function, if the body of the function executes the command data.append('F'), the new entry is added to the end of the list identiﬁed as data within the function, which is one and the same as the list known to the caller as grades. As an aside, we note that reassigning a new value to a formal parameter with a function body, such as by setting data = [], does not alter the actual parameter; such a reassignment simply breaks the alias.

＊

Our hypothetical example of a count method that appends a new element to a list lacks common sense. There is no reason to expect such a behavior, and it would be quite a poor design to have such an unexpected effect on the parameter. There are, however, many legitimate cases in which a function may be designed (and clearly documented) to modify the state of a parameter. As a concrete example, we present the following implementation of a method named scale that’s primary purpose is to multiply all entries of a numeric data set by a given factor.

＊

**def** scale(data, factor):

**for** j **in** range(len(data)):

data[j] \*= factor

**Default Parameter Values**

＊

Python provides means for functions to support more than one possible calling signature. Such a function is said to be ***polymorphic*** (which is Greek for “many forms”). Most notably, functions can declare one or more default values for parameters, thereby allowing the caller to invoke a function with varying numbers of actual parameters. As an artiﬁcial example, if a function is declared with signature

＊

**def** foo(a, b=15, c=27):

there are three parameters, the last two of which offer default values. A caller is welcome to send three actual parameters, as in foo(4, 12, 8), in which case the de- fault values are not used. If, on the other hand, the caller only sends one parameter, foo(4), the function will execute with parameters values a=4, b=15, c=27. If a caller sends two parameters, they are assumed to be the ﬁrst two, with the third be- ing the default. Thus, foo(8, 20) executes with a=8, b=20, c=27. However, it is illegal to deﬁne a function with a signature such as bar(a, b=15, c) with b having a default value, yet not the subsequent c; if a default parameter value is present for one parameter, it must be present for all further parameters.

＊

As a more motivating example for the use of a default parameter, we revisit the task of computing a student’s GPA (see Code Fragment 1.1). Rather than assume direct input and output with the console, we prefer to design a function that computes and returns a GPA. Our original implementation uses a ﬁxed mapping from each letter grade (such as a B−) to a corresponding point value (such as 2.67). While that point system is somewhat common, it may not agree with the system used by all schools. (For example, some may assign an A+ grade a value higher than 4.0.) Therefore, we design a compute gpa function, given in Code Fragment 1.2, which allows the caller to specify a custom mapping from grades to values, while offering the standard point system as a default.

＊



As an additional example of an interesting polymorphic function, we consider Python’s support for range. (Technically, this is a constructor for the range class, but for the sake of this discussion, we can treat it as a pure function.) Three calling syntaxes are supported. The one-parameter form, range(n), generates a sequence of integers from 0 up to but not including n. A two-parameter form, range(start,stop) generates integers from start up to, but not including, stop. A three-parameter form, range(start, stop, step), generates a similar range as range(start, stop), but with increments of size step rather than 1.

＊

This combination of forms seems to violate the rules for default parameters. In particular, when a single parameter is sent, as in range(n), it serves as the stop value (which is the second parameter); the value of start is effectively 0 in that case. However, this effect can be achieved with some sleight of hand, as follows:

＊

**def** range(start, stop=**None**, step=1):

**if** stop **is** **None**:

stop = start

start = 0

...

From a technical perspective, when range(n) is invoked, the actual parameter n will be assigned to formal parameter start. Within the body, if only one parameter is received, the start and stop values are reassigned to provide the desired semantics.

＊

**Keyword Parameters**

＊

The traditional mechanism for matching the actual parameters sent by a caller, to the formal parameters declared by the function signature is based on the concept of ***positional arguments***. For example, with signature foo(a=10, b=20, c=30), parameters sent by the caller are matched, in the given order, to the formal parameters. An invocation of foo(5) indicates that a=5, while b and c are assigned their default values.

＊

Python supports an alternate mechanism for sending a parameter to a function known as a ***keyword argument***. A keyword argument is speciﬁed by explicitly assigning an actual parameter to a formal parameter by name. For example, with the above deﬁnition of function foo, a call foo(c=5) will invoke the function with parameters a=10, b=20, c=5.

＊

A function’s author can require that certain parameters be sent only through the keyword-argument syntax. We never place such a restriction in our own function deﬁnitions, but we will see several important uses of keyword-only parameters in Python’s standard libraries. As an example, the built-in max function accepts a keyword parameter, coincidentally named key, that can be used to vary the notion of “maximum” that is used.

＊

By default, max operates based upon the natural order of elements according to the < operator for that type. But the maximum can be computed by comparing some other aspect of the elements. This is done by providing an auxiliary function that converts a natural element to some other value for the sake of comparison. For example, if we are interested in ﬁnding a numeric value with magnitude that is maximal (i.e., considering −35 to be larger than +20), we can use the calling syntax max(a, b, key=abs). In this case, the built-in abs function is itself sent as the value associated with the keyword parameter key. (Functions are ﬁrst-class objects in Python; see Section 1.10.) When max is called in this way, it will compare abs(a) to abs(b), rather than a to b. The motivation for the keyword syntax as an alternate to positional arguments is important in the case of max. This function is polymorphic in the number of arguments, allowing a call such as max(a,b,c,d); therefore, it is not possible to designate a key function as a traditional positional element. Sorting functions in Python also support a similar key parameter for indicating a nonstandard order. (We explore this further in Section 9.4 and in Section 12.6.1, when discussing sorting algorithms).

＊

1.5.2 Python’s Built-In Functions

＊

Table 1.4 provides an overview of common functions that are automatically avail- able in Python, including the previously discussed abs, max, and range. When choosing names for the parameters, we use identiﬁers x, y, z for arbitrary numeric types, k for an integer, and a, b, and c for arbitrary comparable types. We use the identiﬁer, iterable, to represent an instance of any iterable type (e.g., str, list, tuple, set, dict); we will discuss iterators and iterable data types in Section 1.8. A sequence represents a more narrow category of indexable classes, including str, list, and tuple, but neither set nor dict. Most of the entries in Table 1.4 can be categorized according to their functionality as follows:

＊

**Input/Output**: print, input, and open will be more fully explained in Section 1.6.

＊

**Character Encoding**: ord and chr relate characters and their integer code points. For example, ord( A ) is 65 and chr(65) is A .

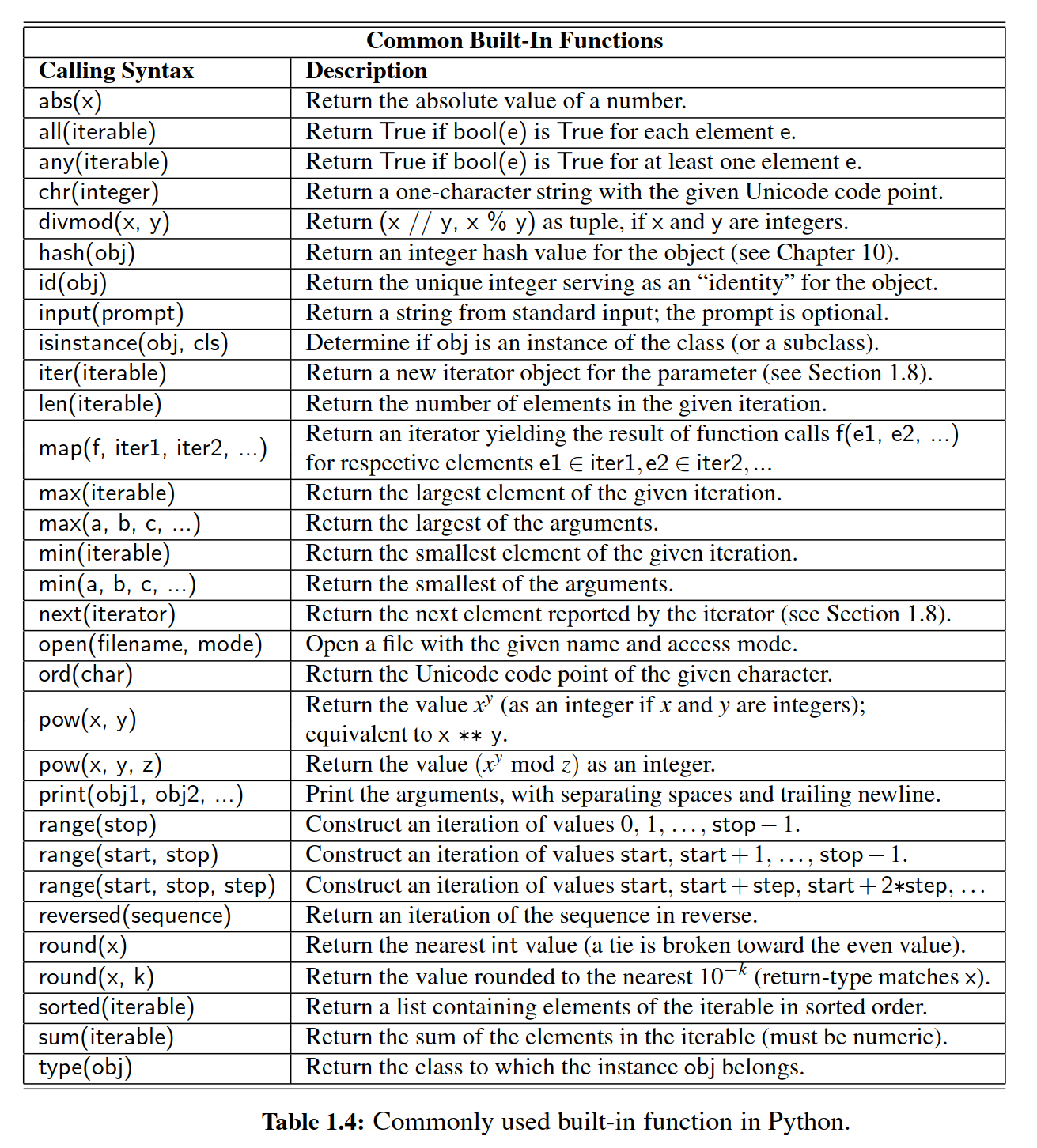
＊

**Mathematics**: abs, divmod, pow, round, and sum provide common mathematical functionality; an additional math module will be introduced in Section 1.11.

＊

**Ordering**: max and min apply to any data type that supports a notion of comparison, or to any collection of such values. Likewise, sorted can be used to produce an ordered list of elements drawn from any existing collection.

**Collections/Iterations**: range generates a new sequence of numbers; len reports the length of any existing collection; functions reversed, all, any, and map operate on arbitrary iterations as well; iter and next provide a general framework for iteration through elements of a collection, and are discussed in Section 1.8.



＊

1.6 Simple Input and Output

＊

In this section, we address the basics of input and output in Python, describing standard input and output through the user console, and Python’s support for reading and writing text ﬁles.

＊

1.6.1 Console Input and Output

＊

**The print Function**

＊

The built-in function, print, is used to generate standard output to the console. In its simplest form, it prints an arbitrary sequence of arguments, separated by spaces, and followed by a trailing newline character. For example, the command print( 'maroon' , 5) outputs the string maroon 5\n . Note that arguments need not be string instances. A nonstring argument x will be displayed as str(x). Without any arguments, the command print() outputs a single newline character.

＊

The print function can be customized through the use of the following keyword parameters (see Section 1.5 for a discussion of keyword parameters):

＊

* By default, the print function inserts a separating space into the output be- tween each pair of arguments. The separator can be customized by providing a desired separating string as a keyword parameter, sep. For example, colon separated output can be produced as print(a, b, c, sep = : ). The separating string need not be a single character; it can be a longer string, and it can be the empty string, sep = ' ', causing successive arguments to be directly con- catenated.  
  ＊
* By default, a trailing newline is output after the ﬁnal argument. An alternative trailing string can be designated using a keyword parameter, end. Designating the empty string end = ' ' suppresses all trailing characters.  
  ＊
* By default, the print function sends its output to the standard console. However, output can be directed to a ﬁle by indicating an output ﬁle stream (see Section 1.6.2) using ﬁle as a keyword parameter.  
  ＊

**The input Function**

＊

The primary means for acquiring information from the user console is a built-in function named input. This function displays a prompt, if given as an optional parameter, and then waits until the user enters some sequence of characters followed by the return key. The formal return value of the function is the string of characters that were entered strictly before the return key (i.e., no newline character exists in the returned string).

＊

When reading a numeric value from the user, a programmer must use the input function to get the string of characters, and then use the int or ﬂoat syntax to construct the numeric value that character string represents. That is, if a call to response = input() reports that the user entered the characters, 2013 , the syntax int(response) could be used to produce the integer value 2013. It is quite common to combine these operations with a syntax such as

＊

year = **int**(input( 'In what year were you born? '))

if we assume that the user will enter an appropriate response. (In Section 1.7 we discuss error handling in such a situation.)

＊

Because input returns a string as its result, use of that function can be combined with the existing functionality of the string class, as described in Appendix A. For example, if the user enters multiple pieces of information on the same line, it is common to call the split method on the result, as in

＊

reply = input( 'Enter x and y, separated by spaces: ')

pieces = reply.split( ) # returns a list of strings, as separated by spaces

x = **ﬂoat**(pieces[0])

y = **ﬂoat**(pieces[1])

**A Sample Program**

＊

Here is a simple, but complete, program that demonstrates the use of the input and print functions. The tools for formatting the ﬁnal output is discussed in Appendix A.

＊

age = **int**(input( 'Enter your age in years: '))

max\_heart\_rate = 206.9 − (0.67 \* age)

# as per Med Sci Sports Exerc.

target = 0.65 \* max\_heart\_rate

print( 'Your target fat-burning heart rate is' , target)

1.6.2 Files

＊

Files are typically accessed in Python beginning with a call to a built-in function, named open, that returns a proxy for interactions with the underlying ﬁle. For example, the command, fp = open( 'sample.txt' ), attempts to open a ﬁle named sample.txt, returning a proxy that allows read-only access to the text ﬁle.

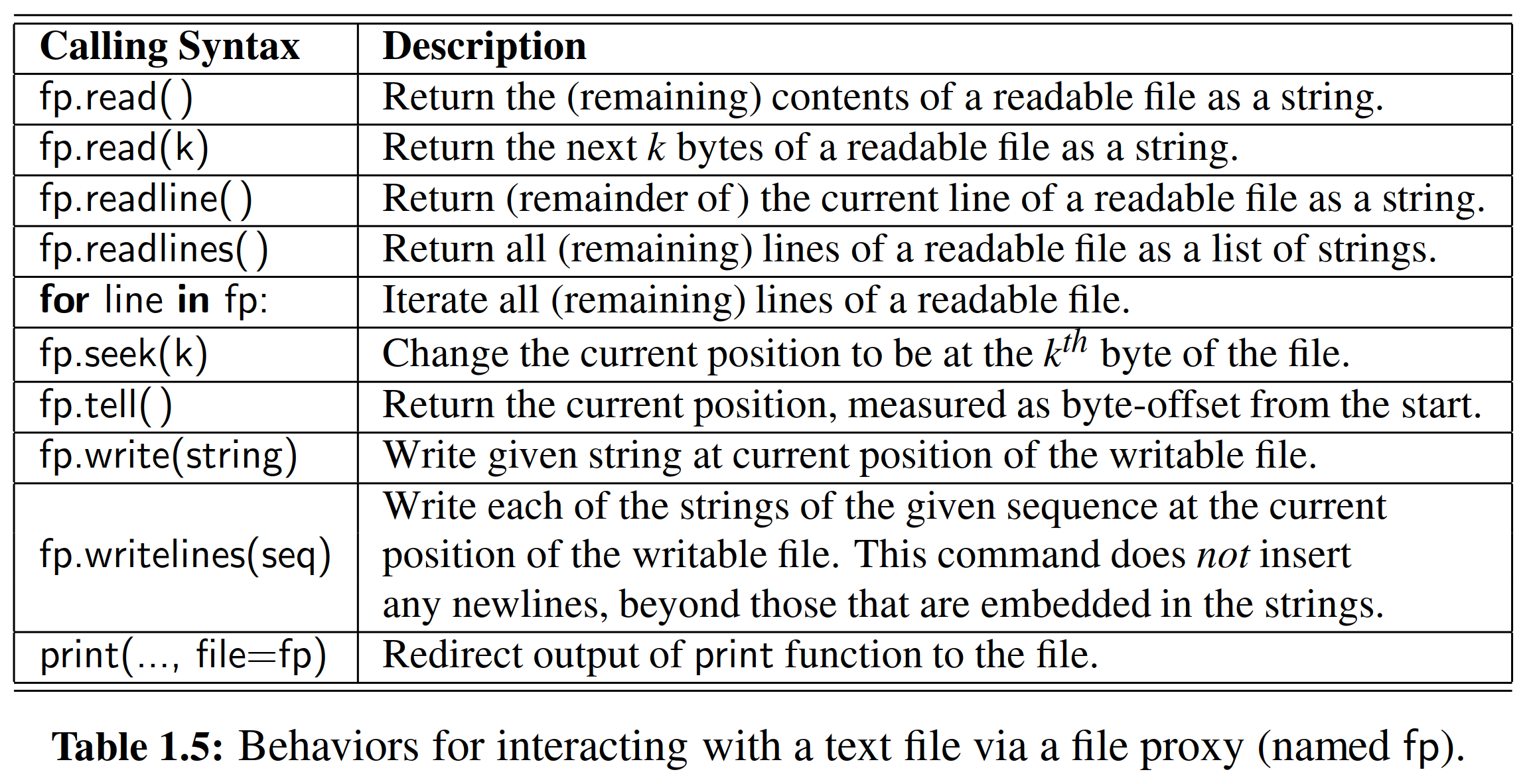
＊

The open function accepts an optional second parameter that determines the access mode. The default mode is 'r' for reading. Other common modes are 'w' for writing to the ﬁle (causing any existing ﬁle with that name to be overwritten), or 'a' for appending to the end of an existing ﬁle. Although we focus on use of text ﬁles, it is possible to work with binary ﬁles, using access modes such as 'rb' or 'wb'.

＊

When processing a ﬁle, the proxy maintains a current position within the ﬁle as an offset from the beginning, measured in number of bytes. When opening a ﬁle with mode 'r' or 'w' , the position is initially 0; if opened in append mode, 'a' , the position is initially at the end of the ﬁle. The syntax fp.close() closes the ﬁle associated with proxy fp, ensuring that any written contents are saved. A summary of methods for reading and writing a ﬁle is given in Table 1.5

＊



**Reading from a File**

＊

The most basic command for reading via a proxy is the read method. When invoked on ﬁle proxy fp, as fp.read(k), the command returns a string representing the next k bytes of the ﬁle, starting at the current position. Without a parameter, the syntax fp.read() returns the remaining contents of the ﬁle in entirety. For convenience, ﬁles can be read a line at a time, using the readline method to read one line, or the readlines method to return a list of all remaining lines. Files also support the for-loop syntax, with iteration being line by line (e.g., **for** line **in** fp:).

＊

**Writing to a File**

＊

When a ﬁle proxy is writable, for example, if created with access mode 'w' or 'a' , text can be written using methods write or writelines. For example, if we deﬁne fp = open( 'results.txt' , w ), the syntax fp.write( 'Hello World.\n' ) writes a single line to the ﬁle with the given string. Note well that write does not explicitly add a trailing newline, so desired newline characters must be embedded directly in the string parameter. Recall that the output of the print method can be redirected to a ﬁle using a keyword parameter, as described in Section 1.6.

＊

1.7 Exception Handling

＊1.7节 异常处理

Exceptions are unexpected events that occur during the execution of a program. An exception might result from a logical error or an unanticipated situation. In Python, ***exceptions*** (also known as ***errors***) are objects that are ***raised*** (or ***thrown***) by code that encounters an unexpected circumstance. The Python interpreter can also raise an exception should it encounter an unexcepted condition, like running out of memory. A raised error may be caught by a surrounding context that “handles” the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program to report an appropriate message to the console.

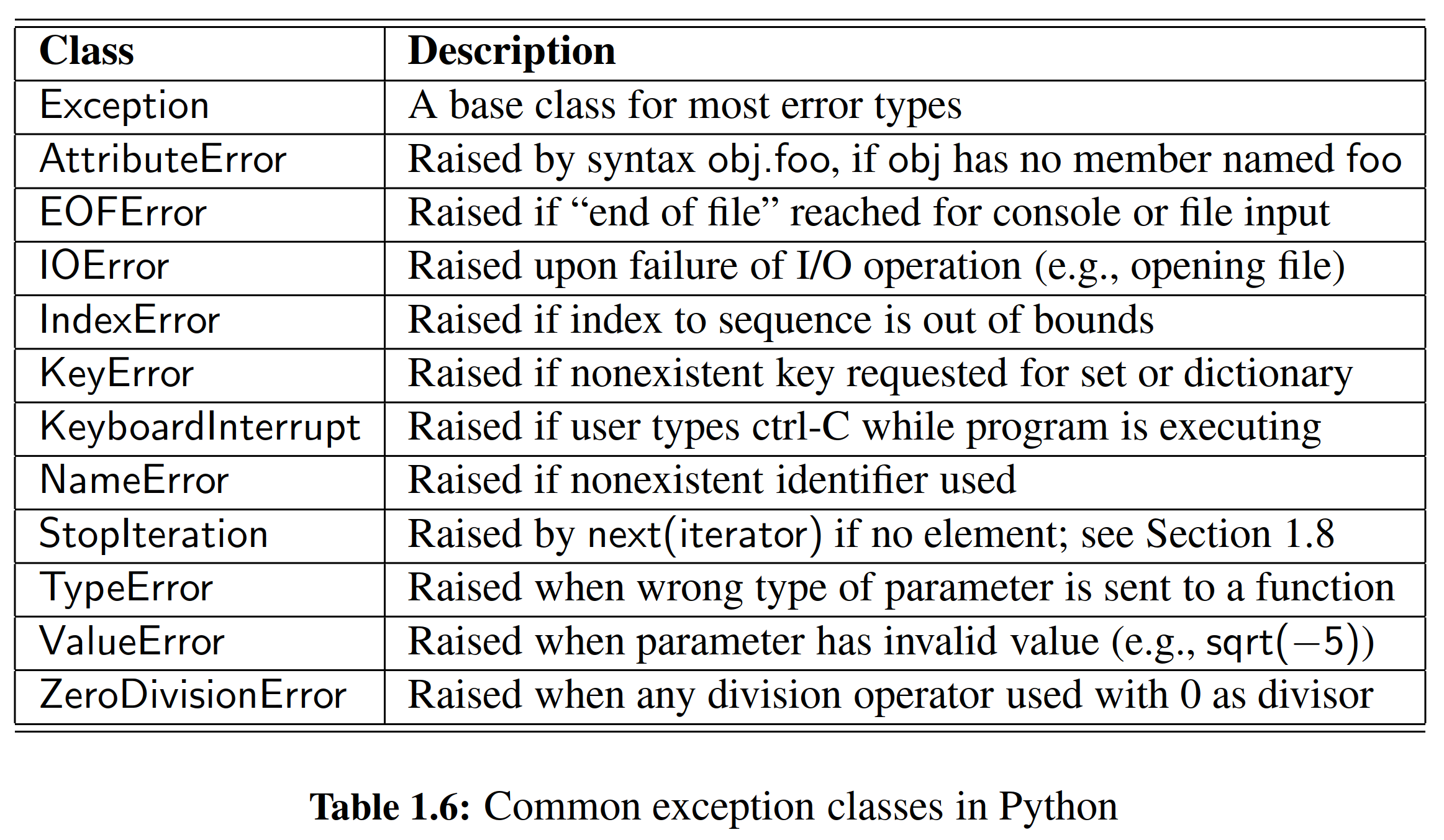
＊我们把程序执行过程中发生的未预料到的事件称为“异常”。能导致异常的原因有很多，比如逻辑错误或者未预料到的情形。在Python语言中，异常（即C语言中与之相同的error）是以对象的形式、被发生了不可预知情形的代码抛出的。当遇到像内存溢出一样的情形时，Python的解释器也能抛出异常。当一个错误被抛出，它可以被周围的上下文捕获并以恰当的方式处理。如果异常未能被捕获，那么这将会导致程序停止运行并且向控制台提交相关信息。

**Common Exception Types**

＊

Python includes a rich hierarchy of exception classes that designate various categories of errors; Table 1.6 shows many of those classes. The Exception class serves as a base class for most other error types. An instance of the various subclasses encodes details about a problem that has occurred. Several of these errors may be raised in exceptional cases by behaviors introduced in this chapter. For example, use of an undeﬁned identiﬁer in an expression causes a NameError, and errant use of the dot notation, as in foo.bar(), will generate an AttributeError if object foo does not support a member named bar.

＊



Sending the wrong number, type, or value of parameters to a function is another common cause for an exception. For example, a call to abs( 'hello' ) will raise a TypeError because the parameter is not numeric, and a call to abs(3, 5) will raise a TypeError because one parameter is expected. A ValueError is typically raised when the correct number and type of parameters are sent, but a value is illegitimate for the context of the function. For example, the int constructor accepts a string, as with int( '137' ), but a ValueError is raised if that string does not represent an integer, as with int( '3.14' ) or int( 'hello' ).

＊

Python’s sequence types (e.g., list, tuple, and str) raise an IndexError when syntax such as data[k] is used with an integer k that is not a valid index for the given sequence (as described in Section 1.2.3). Sets and dictionaries raise a KeyError when an attempt is made to access a nonexistent element.

＊

1.7.1 Raising an Exception

＊

An exception is thrown by executing the **raise** statement, with an appropriate in- stance of an exception class as an argument that designates the problem. For example, if a function for computing a square root is sent a negative value as a parameter, it can raise an exception with the command:

＊

**raise** ValueError( 'x cannot be negative' )

This syntax raises a newly created instance of the ValueError class, with the error message serving as a parameter to the constructor. If this exception is not caught within the body of the function, the execution of the function immediately ceases and the exception is propagated to the calling context (and possibly beyond).

＊

When checking the validity of parameters sent to a function, it is customary to ﬁrst verify that a parameter is of an appropriate type, and then to verify that it has an appropriate value. For example, the sqrt function in Python’s math library performs error-checking that might be implemented as follows:

＊

**def** sqrt(x):

**if** **not** isinstance(x, (int, ﬂoat)):

**raise** TypeError( x must be numeric )

**elif** x < 0:

**raise** ValueError( x cannot be negative )

# do the real work here...

Checking the type of an object can be performed at run-time using the built-in function, isinstance. In simplest form, isinstance(obj, cls) returns True if object, obj, is an instance of class, cls, or any subclass of that type. In the above example, a more general form is used with a tuple of allowable types indicated with the second parameter. After conﬁrming that the parameter is numeric, the function enforces an expectation that the number be nonnegative, raising a ValueError otherwise.

＊

How much error-checking to perform within a function is a matter of debate. Checking the type and value of each parameter demands additional execution time and, if taken to an extreme, seems counter to the nature of Python. Consider the built-in sum function, which computes a sum of a collection of numbers. An implementation with rigorous error-checking might be written as follows:

＊

**def** sum(values):

**if** **not** isinstance(values, collections.Iterable):

**raise** TypeError( parameter must be an iterable type )

total = 0

**for** v **in** values:

**if** not isinstance(v, (int, ﬂoat)):

**raise** TypeError( elements must be numeric )

total = total+ v

**return** total

The abstract base class, collections.Iterable, includes all of Python’s iterable containers types that guarantee support for the for-loop syntax (e.g., list, tuple, set); we discuss iterables in Section 1.8, and the use of modules, such as collections, in Section 1.11. Within the body of the for loop, each element is veriﬁed as numeric before being added to the total. A far more direct and clear implementation of this function can be written as follows:

def sum(values): total = 0

for v in values:

total = total + v

return total

Interestingly, this simple implementation performs exactly like Python’s built-in version of the function. Even without the explicit checks, appropriate exceptions are raised naturally by the code. In particular, if values is not an iterable type, the attempt to use the for-loop syntax raises a TypeError reporting that the object is not iterable. In the case when a user sends an iterable type that includes a nonnumerical element, such as sum([3.14, oops ]), a TypeError is naturally raised by the evaluation of expression total + v. The error message

＊

unsupported operand type(s) for +: ’float’ and ’str’

should be sufﬁciently informative to the caller. Perhaps slightly less obvious is the error that results from sum([ 'alpha' , 'beta' ]). It will technically report a failed attempt to add an int and str, due to the initial evaluation of total + 'alpha' , when total has been initialized to 0.

＊

In the remainder of this book, we tend to favor the simpler implementations in the interest of clean presentation, performing minimal error-checking in most situations.

＊

＊

1.7.2 Catching an Exception

＊1.7.2节 捕获异常

There are several philosophies regarding how to cope with possible exceptional cases when writing code. For example, if a division x/y is to be computed, there is clear risk that a ZeroDivisionError will be raised when variable y has value 0. In an ideal situation, the logic of the program may dictate that y has a nonzero value, thereby removing the concern for error. However, for more complex code, or in a case where the value of y depends on some external input to the program, there remains some possibility of an error.

＊

One philosophy for managing exceptional cases is to ***“look before you leap.”*** The goal is to entirely avoid the possibility of an exception being raised through the use of a proactive conditional test. Revisiting our division example, we might avoid the offending situation by writing:

＊

**if** y != 0:

ratio = x / y

**else**:

... do something else ...

A second philosophy, often embraced by Python programmers, is that “***it is easier to ask for forgiveness than it is to get permission.***” This quote is attributed to Grace Hopper, an early pioneer in computer science. The sentiment is that we need not spend extra execution time safeguarding against every possible exceptional case, as long as there is a mechanism for coping with a problem after it arises. In Python, this philosophy is implemented using a ***try-except*** control structure. Revising our ﬁrst example, the division operation can be guarded as follows:

＊

**try**:

ratio = x / y

**except** ZeroDivisionError:

... do something else ...

In this structure, the “try” block is the primary code to be executed. Although it is a single command in this example, it can more generally be a larger block of indented code. Following the try-block are one or more “except” cases, each with an identiﬁed error type and an indented block of code that should be executed if the designated error is raised within the try-block.

＊

The relative advantage of using a try-except structure is that the non-exceptional case runs efﬁciently, without extraneous checks for the exceptional condition. However, handling the exceptional case requires slightly more time when using a try- except structure than with a standard conditional statement. For this reason, the try-except clause is best used when there is reason to believe that the exceptional case is relatively unlikely, or when it is prohibitively expensive to proactively evaluate a condition to avoid the exception.

＊

Exception handling is particularly useful when working with user input, or when reading from or writing to ﬁles, because such interactions are inherently less predictable. In Section 1.6.2, we suggest the syntax, fp = open( sample.txt ), for opening a ﬁle with read access. That command may raise an IOEeror for a variety of reasons, such as a non-existent ﬁle, or lack of sufﬁcient privilege for opening a ﬁle. It is signiﬁcantly easier to attempt the command and catch the resulting error than it is to accurately predict whether the command will succeed.

＊

We continue by demonstrating a few other forms of the try-except syntax. Exceptions are objects that can be examined when caught. To do so, an identiﬁer must be established with a syntax as follows:

＊

**try**:

fp = open( sample.txt )

**except** IOError **as** e:

print( Unable to open the file: , e)

In this case, the name, e, denotes the instance of the exception that was thrown, and printing it causes a detailed error message to be displayed (e.g., “ﬁle not found”).

＊

A try-statement may handle more than one type of exception. For example, consider the following command from Section 1.6.1:

＊

age = **int**(input( 'Enter your age in years: '))

This command could fail for a variety of reasons. The call to input will raise an EOFError if the console input fails. If the call to input completes successfully, the int constructor raises a ValueError if the user has not entered characters representing a valid integer. If we want to handle two or more types of errors in the same way, we can use a single except-statement, as in the following example:

＊

age = −1 # an initially invalid choice

**while** age <= 0:

**try**:

age = **int**(input( 'Enter your age in years: '))

**if** age <= 0:

print( 'Your age must be positive )

**except** (ValueError, EOFError'):

print( 'Invalid response )

We use the tuple, (ValueError, EOFError), to designate the types of errors that we wish to catch with the except-clause. In this implementation, we catch either error, print a response, and continue with another pass of the enclosing while loop. We note that when an error is raised within the try-block, the remainder of that body is immediately skipped. In this example, if the exception arises within the call to input, or the subsequent call to the int constructor, the assignment to age never occurs, nor the message about needing a positive value. Because the value of age will be unchanged, the while loop will continue. If we preferred to have the while loop continue without printing the 'Invalid response' message, we could have written the exception-clause as

＊

**except** (ValueError, EOFError):

**pass**

The keyword, **pass**, is a statement that does nothing, yet it can serve syntactically as a body of a control structure. In this way, we quietly catch the exception, thereby allowing the surrounding while loop to continue.

＊

In order to provide different responses to different types of errors, we may use two or more except-clauses as part of a try-structure. In our previous example, an EOFError suggests a more insurmountable error than simply an errant value being entered. In that case, we might wish to provide a more speciﬁc error message, or perhaps to allow the exception to interrupt the loop and be propagated to a higher context. We could implement such behavior as follows:

＊

age = −1 # an initially invalid choice

**while** age <= 0:

**try**:

age = int(input( 'Enter your age in years: '))

if age <= 0:

print( Your age must be positive )

**except** ValueError:

print( That is an invalid age specification )

**except** EOFError:

print( There was an unexpected error reading input. )

**raise** # let s re-raise this exception

In this implementation, we have separate except-clauses for the ValueError and EOFError cases. The body of the clause for handling an EOFError relies on another technique in Python. It uses the raise statement without any subsequent argument, to re-raise the same exception that is currently being handled. This allows us to provide our own response to the exception, and then to interrupt the while loop and propagate the exception upward.

＊

In closing, we note two additional features of try-except structures in Python. It is permissible to have a ﬁnal except-clause without any identiﬁed error types, using syntax **except:**, to catch any other exceptions that occurred. However, this technique should be used sparingly, as it is difﬁcult to suggest how to handle an error of an unknown type. A try-statement can have a **ﬁnally** clause, with a body of code that will always be executed in the standard or exceptional cases, even when an uncaught or re-raised exception occurs. That block is typically used for critical cleanup work, such as closing an open ﬁle.

＊

1.8 Iterators and Generators

＊ 1.8节 迭代器与生成器

In section 1.4.2, we introduced the for-loop syntax beginning as:

＊在1.4.2节中，我们介绍了for循环的语句：

**for** element **in** iterable:

＊for element in iterable:

and we noted that there are many types of objects in Python that qualify as being iterable. Basic container types, such as list, tuple, and set, qualify as iterable types. Furthermore, a string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines. User-defined types may also support iteration. In Python, the mechanism for iteration is based upon the following conventions:

＊我们注意到，Python中有许多类型的对象被认为是可迭代的。基本容器类型，如列表，元组和集合，都可以被定义为可迭代类型。此外，字符串可以产生其字符的迭代，字典可以产生其keys的迭代，并且文件可以产生其行的迭代。用户定义的类型也可以支持迭代。在Python中，迭代的机制基于以下约定

* An ***iterator*** is an object that manages an iteration through a series of values. If variable, i, identifies an iterator object, then each call to the build-in function, next(i), produce a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.  
  ＊迭代器是通过一系列值来实现对迭代对象的管理的。如果变量i代表一个迭代器对象，那么每次调用内建函数next(i)都会从底层产生一个后续元素，直到抛出StopIteration异常来表示没有其他元素。
* An ***iterable*** is an object, obj, that produces an iterator via the syntax iter(obj)  
  ＊可迭代的对象obj，可以通过iter(obj)语句生成一个迭代器对象。

By these definitions, an instance of a list is an iterable, but not itself an iterator. With data = [1, 2, 4, 8], it is not legal to call next(data). However, an iterator object can be produced with syntax, I = iter(data), and then each subsequent call to next(i) will return an element of that list. The for-loop syntax in Python simply automates this process, creating an iterator for the give iterable, and then repeatedly calling for the next element until catching the StopIteration exception.

＊通过这些约定可以看到，列表是可迭代的，但是列表本身并不是一个迭代器。定义列表data = [1, 2, 4, 8]，然后调用next(data)是不合法的。然而，可以使用语法I = iter(data)生成迭代器对象，然后每个后续调用next(i)将返回该列表的元素。Python中的for循环已经比较简单地将这个过程自动化了，先是为可迭代对象创建一个迭代器，然后重复调用下一个元素，直到抛出StopIteration异常。

More generally, it is possible to create multiple iterators based upon the same iterable object, with each iterator maintaining its own state of progress. However, iterators typically maintain their state with indirect reference back to the original collection of elements. For example, calling iter(data) on a list instance produces an instance of the list\_iterator class. That iterator does not store its own copy of the list of elements. Instead, it maintains a current *index* into the original list, representing the next element to be reported. Therefore, if the contents of the original list are modified after the iterator is constructed, but before the iteration is complete, the iterator will be reporting the *updated* contents of the list.

＊更一般地，可以基于一个的可迭代对象而创建多个迭代器，并且每个迭代器都可以保持其自身的进度状态。然而，迭代器通常将间接引用的状态保留在原始的元素集合中。例如，在列表实例上调用iter(data)会生成list\_iterator类的实例。该迭代器不存储其自己的元素列表的副本。相反，它将当前索引植入到原始列表中，借此表示要抛出的下一个元素。因此，如果在构建迭代器之后、但又是在在迭代过程完成之前修改了原始列表的内容，那么迭代器将报告列表的更新内容。

Python also supports functions and classes that produce an implicit iterable series of values, that is, without constructing a data structure to store all of its values at once. For example, the call range(1000000) does not return a list of numbers; it returns a range object that is iterable. This object generates the million values one at a time, and only as needed. Such a ***lazy evaluation*** technique has great advantage. In the case of range, it allows a loop of the form, **for** j **in** range(1000000):, to execute without setting aside memory for storing one million values. Also, if such a loop were to be interrupted in some fashion, no time will have been spent computing unused values of the range.

＊Python还支持产生隐式迭代值的函数和类，也就是说，并不需要构造一个数据结构来存储其所有值。例如，调用语句range(1000000)并不返回数字列表; 它返回一个可迭代的range对象。此对象每次生成一个值，并且仅在需要时生成。这样一个惰性取值的技术有很大的优势。在range这个例子下，它允许循环的形式：for j in range (1000000):中的j去进行循环取值，而不设置存储一百万个值的内存。而且，如果这样一个循环以某种方式中断，那么并没有花费时间去计算未使用的值的范围。

We see lazy evaluation used in many of Python’s libraries. For example, the dictionary class supports methods keys(), values(), and items(), which respectively produce a “view” of all keys, values, or (key, value) pairs within a dictionary. None of these methods produces an explicit list of results. Instead, the views that are produced are iterable objects based upon the actual contents of the dictionary. An explicit list of values from such an iteration can be immediately constructed by calling the list class constructor with the iteration as a parameter. For example, the syntax list(range(1000)) produces a list with value from 0 to 999, while the syntax list(d.values()) produces a list that has elements based upon the current values of dictionary d. We can similarly construct a tuple or set instance based upon a given iterable.

＊我们看到许多Python库中使用了惰性取值。例如，词典支持方法keys()，values()和items()，它们分别产生字典中的key，value或(key,value)二元组。这些方法都不产生实际的结果列表。恰恰相反，生成的是基于字典的实际内容的可迭代对象。可以把迭代器作为列表类的参数，来构造这种迭代的值的显式列表。例如，语句list(range(1000))会产生一个值为0到999的列表，而语句(d.values())则会根据字典d的当前值产生一个包含元素的列表。我们可以基于给定的可迭代对象，来类似地构造元组或生成实例。

**Generators**

In Section 2.3.4, we will explain how to deﬁne a class whose instances serve as iterators. However, the most convenient technique for creating iterators in Python is through the use of generators. A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series. As an example, consider the goal of determining all factors of a positive integer. For example, the number 100 has factors 1, 2, 4, 5, 10, 20, 25, 50, 100. A traditional function might produce and return a list containing all factors, implemented as:

＊在2.3.4节中，我们将要介绍如何定义实例是迭代器的那种类。然而，在Python中创建迭代器的最方便的方式是使用生成器。生成器的语法看起来与函数相似，但是生成器不是返回数值，生成器中的yield语句指示序列中的每个元素。例如，要生成所有一个正整数的所有因数，比如说，数字100具有1，2，4，5，10，20，25，50，100这些因数。传统函数可能产生并返回包含所有因数的列表，代码一般这样写：

**def** factors(n): # traditional function that computes factors

results = [] # store factors in a new list

**for** k **in** range(1,n+1):

**if** n % k == 0: # divides evenly, thus k is a factor

results.append(k) # add k to the list of factors

**return** results # return the entire list

In contrast, an implementation of a *generator* for computing those factors could be implemented as follows:

＊相比之下，用于计算这些因子的生成器可以如下实现：

**def** factors(n): # generator that computes factors

**for** k **in** range(1,n+1):

**if** n % k == 0: # divides evenly, thus k is a factor

**yield** k # yield this factor as next result

Notice use of the keyword **yield** rather than **return** to indicate a result. This indicates to Python that we are deﬁning a generator, rather than a traditional function. It is illegal to combine yield and return statements in the same implementation, other than a zero-argument return statement to cause a generator to end its execution. If a programmer writes a loop such as **for** factor **in** factors(100):, an instance of our generator is created. For each iteration of the loop, Python executes our procedure until a yield statement indicates the next value. At that point, the procedure is temporarily interrupted, only to be resumed when another value is requested. When the ﬂow of control naturally reaches the end of our procedure (or a zero-argument return statement), a StopIteration exception is automatically raised. Although this particular example uses a single yield statement in the source code, a generator can rely on multiple yield statements in different constructs, with the generated series determined by the natural ﬂow of control. For example, we can greatly improve the efﬁciency of our generator for computing factors of a number, n, by only testing values up to the square root of that number, while reporting the factor n//k that is associated with each k (unless n//k equals k). We might implement such a generator as follows:

＊请注意使用关键字yield而不是用return返回结果。这表明了我们定义了一个发生器，而不是一个普通的函数。将yield和return语句组合在相同的实现中是非法的，生成器的结束执行也不是通过零参数的return语句完成的。如果一个程序员写了一个循环，例如for factor in factors(100):，那么就会创建一个生成器。对于循环的每次迭代，Python会一直执行程序语句，如果在执行过程中遇到指示下一个数值的yield语句，进程就停止。此时，进程只是暂时被中断，只有在请求另一个值时才会恢复该过程。当程序运行到了我们的过程结束时（或零参数返回语句时），会自动引发StopIteration异常。尽管该特定示例在源代码中使用单个yield语句，但生成器可以依赖于不同结构中的多个yield语句。例如，我们可以通过将循环控制到n的平方根，而在循环过程中报告每个n//k，这可以大大提高生成器的计算效率。根据这个分析可以写出如下的生成器程序代码：

**def** factors(n): # generator that computes factors

k = 1

**while** k \* k < n: while k < sqrt(n)

**if** n % k == 0:

**yield** k

**yield** n // k

k += 1

**if** k \* k == n: # special case if n is perfect square

**yield** k

We should note that this generator differs from our ﬁrst version in that the factors are not generated in strictly increasing order. For example, factors(100) generates the series 1, 100, 2, 50, 4, 25, 5, 20, 10.

＊我们应该注意到，这种生成器与我们的第一个版本不同，因为这些因数不是以严格的增加顺序生成的。例如，factor(100)产生1，100，2，50，4，25，5，20，10。

In closing, we wish to emphasize the beneﬁts of lazy evaluation when using a generator rather than a traditional function. The results are only computed if requested, and the entire series need not reside in memory at one time. In fact, a generator can effectively produce an inﬁnite series of values. As an example, the Fibonacci numbers form a classic mathematical sequence, starting with value 0, then value 1, and then each subsequent value being the sum of the two preceding values. Hence, the Fibonacci series begins as: 0, 1, 1, 2, 3, 5, 8, 13, .... The following generator produces this inﬁnite series.

＊最后，我们强调一下使用生成器而非一般函数的好处。而好处就是数值只有在需要的时候才会被计算出来，整个序列不需要一次性全部保存在内存中。事实上，生成器可以高效生成一个无穷序列（译者按：这真的是太惊人了！）。例如，斐波纳契数列是一个经典的数学序列，从值0开始，然后是值1，然后每个后续值是前面两个值的和。因此，斐波纳契系列的样式为：0，1，1，2，3，5，8，13，...。下面的生成器可以生成这个无穷的数列。

**def** ﬁbonacci( ):

a = 0

b = 1

**while** **True**: # keep going...

**yield** a # report value, a, during this pass

future = a + b

a = b # this will be next value reported

b = future # and subsequently this

1.9 Additional Python Convenience

＊

In this section, we introduce several features of Python that are particularly convenient for writing clean, concise code. Each of these syntaxes provide functionality that could otherwise be accomplished using functionality that we have introduced earlier in this chapter. However, at times, the new syntax is a more clear and direct expression of the logic.

＊

1.9.1 Conditional

＊

Python supports a conditional expression syntax that can replace a simple control structure. The general syntax is an expression of the form:

＊

*expr1* **if** *condition* **else** *expr2*

This compound expression evaluates to expr1 if the condition is true, and otherwise evaluates to expr2. For those familiar with Java or C++, this is equivalent to the syntax, *condition* ? *expr1* : *expr2*, in those languages.

＊

As an example, consider the goal of sending the absolute value of a variable, n, to a function (and without relying on the built-in abs function, for the sake of ex- ample). Using a traditional control structure, we might accomplish this as follows:

＊

**if** n >= 0:

param = n

**else**:

param = −n

result = foo(param) # call the function

With the conditional expression syntax, we can directly assign a value to variable, param, as follows:

＊

param = n **if** n >= 0 **else** −n # pick the appropriate value result = foo(param) # call the function

In fact, there is no need to assign the compound expression to a variable. A conditional expression can itself serve as a parameter to the function, written as follows:

＊

result = foo(n **if** n >= 0 **else** −n)

Sometimes, the mere shortening of source code is advantageous because it avoids the distraction of a more cumbersome control structure. However, we recommend that a conditional expression be used only when it improves the readability of the source code, and when the ﬁrst of the two options is the more “natural” case, given its prominence in the syntax. (We prefer to view the alternative value as more exceptional.)

＊

1.9.2 Comprehension Syntax

＊

A very common programming task is to produce one series of values based upon the processing of another series. Often, this task can be accomplished quite simply in Python using what is known as a ***comprehension syntax***. We begin by demonstrating ***list comprehension***, as this was the ﬁrst form to be supported by Python. Its general form is as follows:

＊

[ *expression* **for** *value* **in** *iterable* if *condition* ]

We note that both *expression* and *condition* may depend on *value*, and that the if-clause is optional. The evaluation of the comprehension is logically equivalent to the following traditional control structure for computing a resulting list:

＊

result = [ ]

**for** value **in** iterable:

**if** condition:

result.append(expression)

As a concrete example, a list of the squares of the numbers from 1 to n, that is [1, 4, 9, 16, 25,... , *n*2], can be created by traditional means as follows:

＊

squares = [ ]

**for** k **in** range(1, n+1):

squares.append(k \* k)

With list comprehension, this logic is expressed as follows:

＊

squares = [k \* k **for** k **in** range(1, n+1)]

As a second example, Section 1.8 introduced the goal of producing a list of factors for an integer n. That task is accomplished with the following list comprehension:

＊

factors = [k **for** k **in** range(1,n+1) **if** n % k == 0]

Python supports similar comprehension syntaxes that respectively produce a set, generator, or dictionary. We compare those syntaxes using our example for producing the squares of numbers.

＊

[ k \* k **for** k **in** range(1, n+1) ] list comprehension

{ k \* k **for** k **in** range(1, n+1) } set comprehension

( k \* k **for** k **in** range(1, n+1) ) generator comprehension

{ k : k \* k **for** k **in** range(1, n+1) } dictionary comprehension

The generator syntax is particularly attractive when results do not need to be stored in memory. For example, to compute the sum of the ﬁrst n squares, the genera- tor syntax, total = sum(k \* k **for** k **in** range(1, n+1)), is preferred to the use of an explicitly instantiated list comprehension as the parameter.

＊

1.9.3 Packing and Unpacking of Sequence

＊

Python provides two additional conveniences involving the treatment of tuples and other sequence types. The ﬁrst is rather cosmetic. If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided. For example, the assignment

＊

data = 2, 4, 6, 8

results in identiﬁer, data, being assigned to the tuple (2, 4, 6, 8). This behavior is called ***automatic packing*** of a tuple. One common use of packing in Python is when returning multiple values from a function. If the body of a function executes the command,

＊

return x, y

it will be formally returning a single object that is the tuple (x, y).

＊

As a dual to the packing behavior, Python can automatically ***unpack*** a sequence, allowing one to assign a series of individual identiﬁers to the elements of sequence. As an example, we can write

＊

a, b, c, d = range(7, 11)

which has the effect of assigning a=7, b=8, c=9, and d=10, as those are the four values in the sequence returned by the call to range. For this syntax, the right-hand side expression can be any *iterable* type, as long as the number of variables on the left-hand side is the same as the number of elements in the iteration.

＊

This technique can be used to unpack tuples returned by a function. For example, the built-in function, divmod(a, b), returns the pair of values (a // b, a % b) associated with an integer division. Although the caller can consider the return value to be a single tuple, it is possible to write

＊

quotient, remainder = divmod(a, b)

to separately identify the two entries of the returned tuple. This syntax can also be used in the context of a for loop, when iterating over a sequence of iterables, as in

＊

**for** x, y **in** [ (7, 2), (5, 8), (6, 4) ]:

In this example, there will be three iterations of the loop. During the ﬁrst pass, x=7 and y=2, and so on. This style of loop is quite commonly used to iterate through key-value pairs that are returned by the items() method of the dict class, as in:

＊

**for** k, v **in** mapping.items():

**Simultaneous Assignments**

＊

The combination of automatic packing and unpacking forms a technique known as ***simultaneous assignment***, whereby we explicitly assign a series of values to a series of identiﬁers, using a syntax:

＊

x, y, z = 6, 2, 5

In effect, the right-hand side of this assignment is automatically packed into a tuple, and then automatically unpacked with its elements assigned to the three identiﬁers on the left-hand side.

＊

When using a simultaneous assignment, all of the expressions are evaluated on the right-hand side before any of the assignments are made to the left-hand variables. This is signiﬁcant, as it provides a convenient means for swapping the values associated with two variables:

＊

j, k = k, j

With this command, j will be assigned to the old value of k, and k will be assigned to the old value of j. Without simultaneous assignment, a swap typically requires more delicate use of a temporary variable, such as

＊

temp = j

j = k

k = temp

With the simultaneous assignment, the unnamed tuple representing the packed values on the right-hand side implicitly serves as the temporary variable when performing such a swap.

＊

The use of simultaneous assignments can greatly simplify the presentation of code. As an example, we reconsider the generator on page 41 that produces the Fibonacci series. The original code requires separate initialization of variables a and b to begin the series. Within each pass of the loop, the goal was to reassign a and b, respectively, to the values of b and a+b. At the time, we accomplished this with brief use of a third variable. With simultaneous assignments, that generator can be implemented more directly as follows:

＊

**def** ﬁbonacci( ):

a, b = 0, 1

**while** **True**:

**yield** a

a, b = b, a+b

1.10 Scopes and Namespace

＊

When computing a sum with the syntax x + y in Python, the names x and y must have been previously associated with objects that serve as values; a NameError will be raised if no such deﬁnitions are found. The process of determining the value associated with an identiﬁer is known as ***name resolution***.

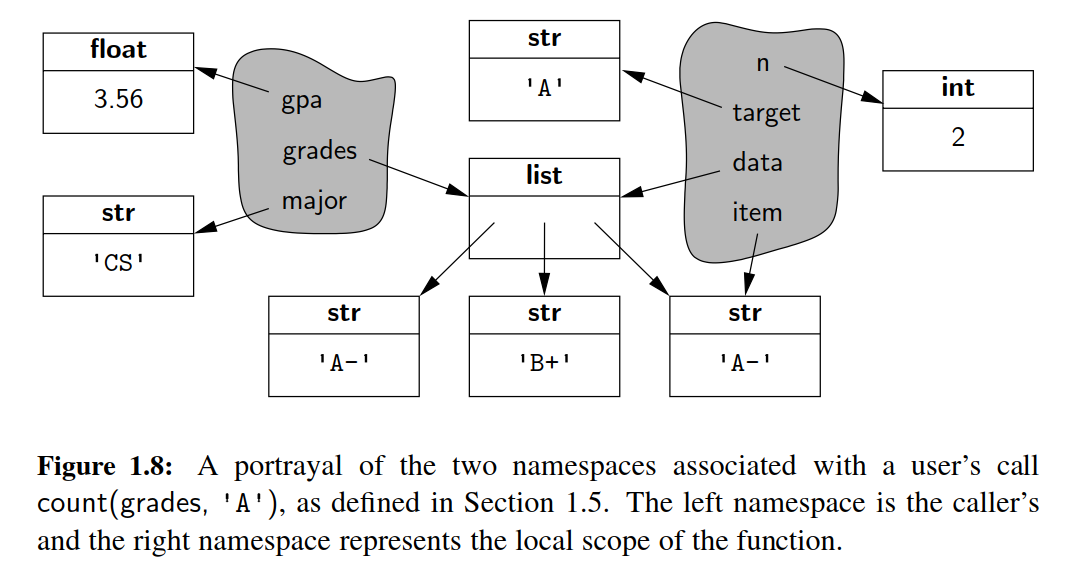
＊

Whenever an identiﬁer is assigned to a value, that deﬁnition is made with a speciﬁc scope. Top-level assignments are typically made in what is known as ***global*** scope. Assignments made within the body of a function typically have scope that is ***local*** to that function call. Therefore, an assignment, x = 5, within a function has no effect on the identiﬁer, x, in the broader scope.

＊

Each distinct scope in Python is represented using an abstraction known as a ***namespace***. A namespace manages all identiﬁers that are currently deﬁned in a given scope. Figure 1.8 portrays two namespaces, one being that of a caller to our count function from Section 1.5, and the other being the local namespace during the execution of that function.

＊



Python implements a namespace with its own dictionary that maps each identifying string (e.g., 'n' ) to its associated value. Python provides several ways to examine a given namespace. The function, dir, reports the names of the identiﬁers in a given namespace (i.e., the keys of the dictionary), while the function, vars, returns the full dictionary. By default, calls to dir() and vars() report on the most locally enclosing namespace in which they are executed.

＊

When an identiﬁer is indicated in a command, Python searches a series of namespaces in the process of name resolution. First, the most locally enclosing scope is searched for a given name. If not found there, the next outer scope is searched, and so on. We will continue our examination of namespaces, in Section 2.5, when discussing Python’s treatment of object-orientation. We will see that each object has its own namespace to store its attributes, and that classes each have a namespace as well.

＊

**First-Class Objects**

＊

In the terminology of programming languages, ***ﬁrst-class objects*** are instances of a type that can be assigned to an identiﬁer, passed as a parameter, or returned by a function. All of the data types we introduced in Section 1.2.3, such as int and list, are clearly ﬁrst-class types in Python. In Python, functions and classes are also treated as ﬁrst-class objects. For example, we could write the following:

＊

scream = print # assign name ’scream’ to the function denoted as ’print’

scream( Hello ) # call that function

In this case, we have not created a new function, we have simply deﬁned scream as an alias for the existing print function. While there is little motivation for precisely this example, it demonstrates the mechanism that is used by Python to al- low one function to be passed as a parameter to another. On page 28, we noted that the built-in function, max, accepts an optional keyword parameter to specify a non-default order when computing a maximum. For example, a caller can use the syntax, max(a, b, key=abs), to determine which value has the larger absolute value. Within the body of that function, the formal parameter, key, is an identiﬁer that will be assigned to the actual parameter, abs.

＊

In terms of namespaces, an assignment such as scream = print, introduces the identiﬁer, scream, into the current namespace, with its value being the object that represents the built-in function, print. The same mechanism is applied when a user-deﬁned function is declared. For example, our count function from Section 1.5 beings with the following syntax:

＊

**def** count(data, target):

...

Such a declaration introduces the identiﬁer, count, into the current namespace, with the value being a function instance representing its implementation. In similar fashion, the name of a newly deﬁned class is associated with a representation of that class as its value. (Class deﬁnitions will be introduced in the next chapter.)

＊

1.11 Modules and the Import Statements

＊

We have already introduced many functions (e.g., max) and classes (e.g., list) that are deﬁned within Python’s built-in namespace. Depending on the version of Python, there are approximately 130–150 deﬁnitions that were deemed signiﬁcant enough to be included in that built-in namespace.

＊

Beyond the built-in deﬁnitions, the standard Python distribution includes per- haps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as ***modules***, that can be ***imported*** from within a pro- gram. As an example, we consider the math module. While the built-in namespace includes a few mathematical functions (e.g., abs, min, max, round), many more are relegated to the math module (e.g., sin, cos, sqrt). That module also deﬁnes approximate values for the mathematical constants, pi and e.

＊

Python’s **import** statement loads deﬁnitions from a module into the current namespace. One form of an import statement uses a syntax such as the following:

＊

**from** math **import** pi, sqrt

This command adds both pi and sqrt, as deﬁned in the math module, into the current namespace, allowing direct use of the identiﬁer, pi, or a call of the function, sqrt(2). If there are many deﬁnitions from the same module to be imported, an asterisk may be used as a wild card, as in, **from** math **import** \* , but this form should be used sparingly. The danger is that some of the names deﬁned in the module may conﬂict with names already in the current namespace (or being imported from another module), and the import causes the new deﬁnitions to replace existing ones.

＊

Another approach that can be used to access many deﬁnitions from the same module is to import the module itself, using a syntax such as:

**import** math

Formally, this adds the identiﬁer, math, to the current namespace, with the module as its value. (Modules are also ﬁrst-class objects in Python.) Once imported, individual deﬁnitions from the module can be accessed using a fully-qualiﬁed name, such as math.pi or math.sqrt(2).

＊

**Creating a New Module**

＊

To create a new module, one simply has to put the relevant deﬁnitions in a ﬁle named with a .py sufﬁx. Those deﬁnitions can be imported from any other .py ﬁle within the same project directory. For example, if we were to put the deﬁnition of our count function (see Section 1.5) into a ﬁle named utility.py, we could import that function using the syntax, **from** utility **import** count.

＊

It is worth noting that top-level commands with the module source code are executed when the module is ﬁrst imported, almost as if the module were its own script. There is a special construct for embedding commands within the module that will be executed if the module is directly invoked as a script, but not when the module is imported from another script. Such commands should be placed in a body of a conditional statement of the following form,

＊

**if** \_\_name\_\_ == '\_\_main\_\_':

Using our hypothetical utility.py module as an example, such commands will be executed if the interpreter is started with a command python utility.py, but not when the utility module is imported into another context. This approach is often used to embed what are known as ***unit tests*** within the module; we will discuss unit testing further in Section 2.2.4.

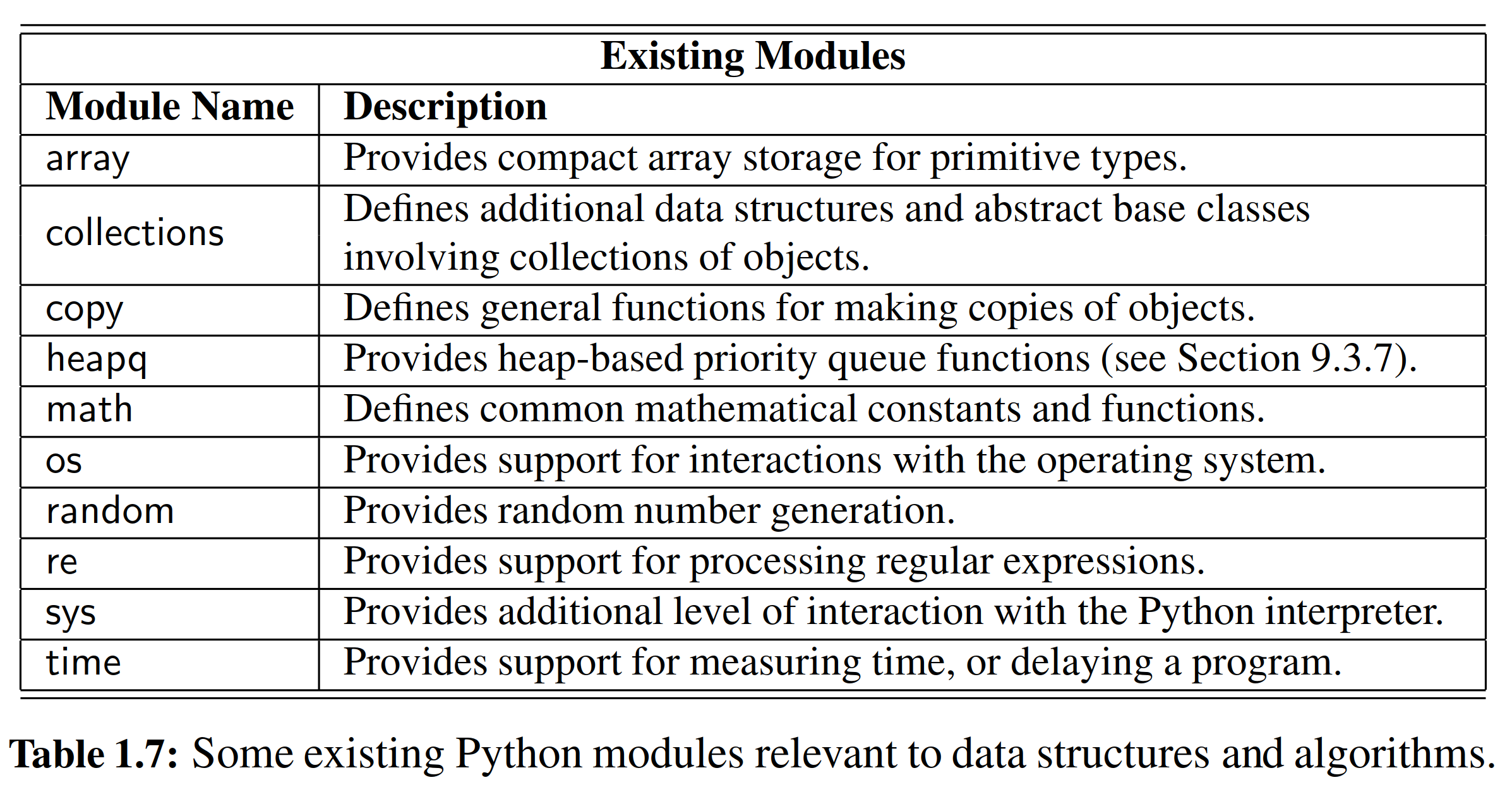
＊

1.11.1 Existing Modules

＊

Table 1.7 provides a summary of a few available modules that are relevant to a study of data structures. We have already discussed the math module brieﬂy. In the remainder of this section, we highlight another module that is particularly important for some of the data structures and algorithms that we will study later in this book.

＊



**Pseudo-Random Number Generation**

＊

Python’s random module provides the ability to generate pseudo-random numbers, that is, numbers that are statistically random (but not necessarily truly random). A ***pseudo-random number generator*** uses a deterministic formula to generate the next number in a sequence based upon one or more past numbers that it has generated. Indeed, a simple yet popular pseudo-random number generator chooses its next number based solely on the most recently chosen number and some additional parameters using the following formula.

＊

next = (a\*current + b) % n;

where a, b, and n are appropriately chosen integers. Python uses a more advanced technique known as a ***Mersenne twister***. It turns out that the sequences generated by these techniques can be proven to be statistically uniform, which is usually good enough for most applications requiring random numbers, such as games. For applications, such as computer security settings, where one needs unpredictable random sequences, this kind of formula should not be used. Instead, one should ideally sample from a source that is actually random, such as radio static coming from outer space.

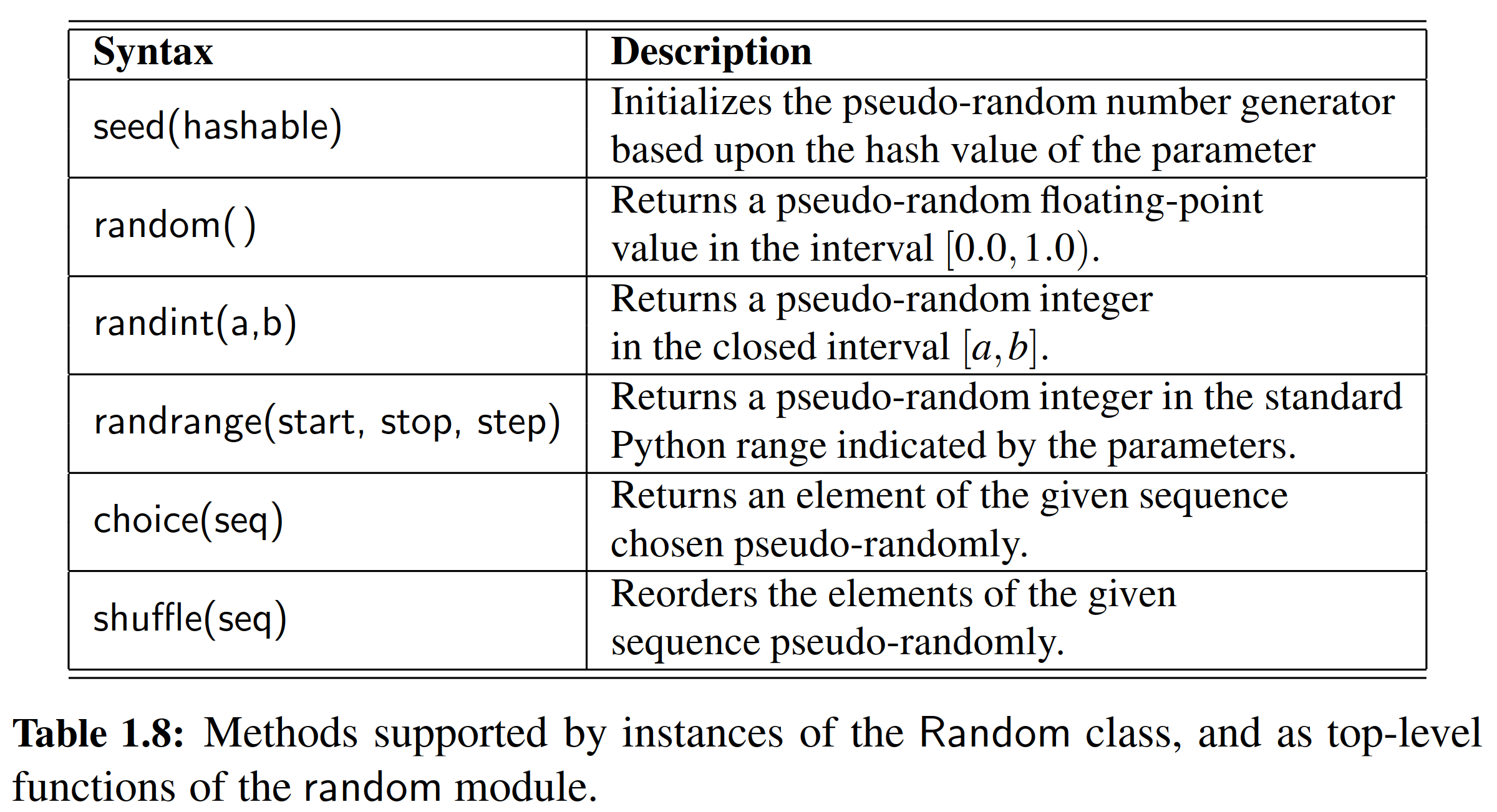
＊

Since the next number in a pseudo-random generator is determined by the previous number(s), such a generator always needs a place to start, which is called its ***seed***. The sequence of numbers generated for a given seed will always be the same. One common trick to get a different sequence each time a program is run is to use a seed that will be different for each run. For example, we could use some timed input from a user or the current system time in milliseconds.

＊

Python’s random module provides support for pseudo-random number generation by deﬁning a Random class; instances of that class serve as generators with independent state. This allows different aspects of a program to rely on their own pseudo-random number generator, so that calls to one generator do not affect the sequence of numbers produced by another. For convenience, all of the methods supported by the Random class are also supported as stand-alone functions of the random module (essentially using a single generator instance for all top-level calls).

＊



**END**

# 六、实验体会

Python是动态的语言，没有预先声明变量的类型。这一点刚从C语言转过来人可能感到很不适应，后来我才知道，Python的内存管理机制与C语言已经完全不一样了，这是因为C语言作为静态语言的代表，是将变量固定在某一内存的区域中，任何数值的变化都在这块区域改变。但是Python不同，它本身是从C语言进行二次开发得到的一门新的解释性语言，它很好地借用了C语言中的指针的概念，在设计Python的过程中，C语言中的malloc函数被广泛使用，从而使得变量的自增或者赋值基本上都会新开辟另外大小的内存片段，从而直接避开了声明数值类型的麻烦，而且旧的区域在赋值或者自增后，将会被free函数清空，从而可以被再次调用。

# 七、参考文献

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, *Data Structures and Algorithms in Python*

[2] 数据结构与算法分析：C语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社

[3] 算法导论（原书第三版），（美）科尔曼（Cormen，T.H.）等；殷建平等译. 北京：机械工业出版社