Module I

Getting Started with Python Programming

- Guido van Rossum invented the Python programming language in the early 1990s.
- Python is a high-level, general-purpose, interpreted programming language for solving problems on modern computer systems.
- The language and many supporting tools are free, and Python programs can run on any operating system.

Running Code in the Interactive Shell

- Python is an **interpreted language,** and you can run simple Python expressions and statements in an interactive programming environment called the **shell.**
- The easiest way to open a Python shell is to launch the **IDLE** (Integrated DeveLopment Environment).
- This is an integrated program development environment that comes with the Python installation. When you do this, a window named Python Shell opens.

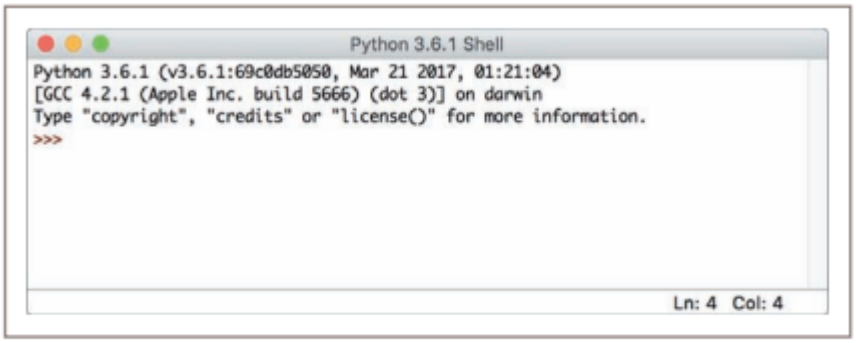

Figure 1-6 Python shell window

- A shell window contains an opening message followed by the special symbol >>>, called a **shell prompt.**
- The cursor at the shell prompt waits for you to enter a Python command.
- Note that you can get immediate help by entering help at the shell prompt or selecting Help from the window's drop-down menu.
- When you enter an expression or statement, Python evaluates it and displays its result, if there is one, followed by a new prompt.

```
>>> 3 + 4                       # Simple arithmetic
7
>>> 3                           # The value of 3 is
3
>>> "Python is really cool!"    # Use a string for text
'Python is really cool!'
>>> name = "Ken Lambert"        # Give a variable a value
>>> name                        # The value of name is
'Ken Lambert'
>>> "Hi there, " + name         # Create some new text
'Hi there, Ken Lambert'
>>> print('Hi there')           # Output some text
Hi there
>>> print("Hi there,", name)    # Output two values
Hi there, Ken Lambert
```

➢

The IDLE programming environment uses color coding to help the reader pick out different elements in the code. In this example, the items within quotation marks are in green, the names of standard functions are in purple, program comments are in red, and the responses of IDLE to user commands are in blue. The remaining code is in black.

- ➢ The Python shell is useful for experimenting with short expressions or statements to learn new features of the language, as well as for consulting documentation on the language.
- ➢ To quit the Python shell, you can either select the window's close box or press the Control1D key combination.

IDLE is Python's Integrated Development and Learning Environment. It allows programmers to easily write Python code. Just like Python Shell, IDLE can be used **to execute a single statement and create, modify, and execute Python scripts**. **Python Shell is a command line tool that starts up the python interpreter**. You can test simple programs and also write some short programs. However, in order to write a more complexed python program you need an editor. IDLE, on the other hand, has combined the above two needs and bundled them as a package. IDLE consists of Python Shell, and Text editor that supports highlights for python grammar and etc.

IPython

IPython (Interactive Python) is a command shell for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history. IPython provides the following features:

- Interactive shells
- A browser-based notebook interface with support for code, text, mathematical expressions, inline plots and other media.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into one's own projects.

- Tools for parallel computing.

IPython is based on an architecture that provides parallel and distributed computing. IPython enables parallel applications to be developed, executed, debugged and monitored interactively, hence the I (Interactive) in IPython.[4] This architecture abstracts out parallelism, enabling IPython to support many different styles of parallelism[5] including:

- Single program, multiple data (SPMD) parallelism
- Multiple program, multiple data (MPMD) parallelism
- Message passing using MPI
- Task parallelism
- Data parallelism
- Combinations of these approaches
- Custom user defined approaches

- The IPython Notebook is now known as the Jupyter Notebook.
- It is an interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media.
- Jupyter Notebook (formerly IPython Notebooks) is **a web-based interactive computational environment for creating, executing, and visualizing Jupyter notebooks**.

Detecting and Correcting Syntax Errors

Programmers inevitably make typographical errors when editing programs, and the Python interpreter will nearly always detect them. Such errors are called **syntax errors.** The term syntax refers to the rules for forming sentences in a language. When Python encounters a syntax error in a program, it halts execution with an error message. The following sessions with the Python shell show several types of syntax errors and the corresponding error messages:

```
>>> length = int(input("Enter the length: "))
Enter the length: 44
>>> print(lenth)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
NameError: name 'lenth' is not defined
```

The first statement assigns an input value to the variable length. The next statement attempts to print the value of the variable lenth. Python responds that this name is not defined. Although the programmer might have meant to write the variable length, Python can read only what the programmer actually entered. This is a good example of the rule that a computer can read only the instructions it receives, not the instructions we intend to give it. The next statement attempts to print the value of the correctly spelled variable. However, Python still generates an error message.

```
>>> print(length)
SyntaxError: unexpected indent
```

In this error message, Python explains that this line of code is unexpectedly indented. In fact, there is an extra space before the word print. Indentation is significant in Python code. Each line of code entered at a shell prompt or in a script must begin in the leftmost column, with no leading spaces. The only exception to this rule occurs in control statements and definitions, where nested statements must be indented one or more spaces.

You might think that it would be painful to keep track of **indentation in** a program. However, in compensation, the Python language is much simpler than other programming languages. Consequently, there are fewer types of syntax errors to encounter and correct, and a lot less syntax for you to learn!

In our final example, the programmer attempts to add two numbers, but forgets to include the second one:

```
>>> 3 +
SyntaxError: invalid syntax
```

How Python Works

Whether you are running Python code as a script or interactively in a shell, the Python interpreter does a great deal of work to carry out the instructions in your program. This work can be broken into a series of steps,

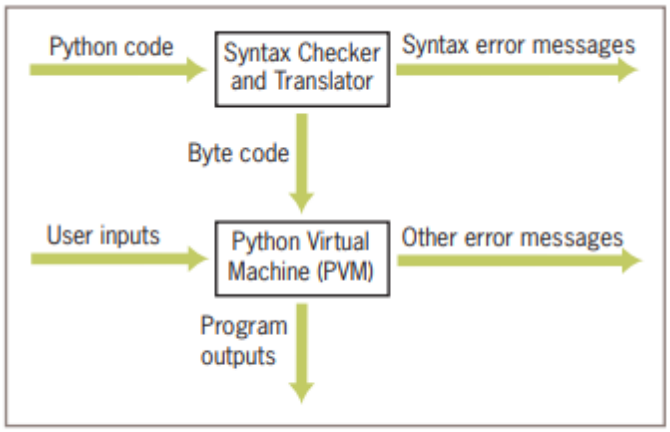

Figure 1-9 Steps in interpreting a Python program

1. The interpreter reads a Python expression or statement, also called the source code, and verifies that it is well formed. In this step, the interpreter behaves like a strict English teacher who rejects any sentence that does not adhere to the grammar rules, or syntax, of the language. As soon as the interpreter encounters such an error, it halts translation with an error message.

2. If a Python expression is well formed, the interpreter then translates it to an equivalent form in a low-level language called byte code. When the interpreter runs a script, it completely translates it to byte code.

3. This byte code is next sent to another software component, called the Python virtual machine (PVM), where it is executed. If another error occurs during this step, execution also halts with an error message.

The Software Development Process

There is much more to programming than writing lines of code, just as there is more to building houses than pounding nails. The "more" consists of organization and planning, and various conventions for diagramming those plans. Computer scientists refer to the process of planning and organizing a program as software development. There are several approaches to software development. One version is known as the waterfall model.

The waterfall model consists of several phases:

1. ***Customer request***—In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution. This step is also called the user requirements phase.

2. ***Analysis***—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.

3. ***Design***—The programmers determine how the program will do its task.

4. ***Implementation***—The programmers write the program. This step is also called the coding phase.

5. ***Integration***—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.

6. ***Maintenance***—Programs usually have a long life; a life span of 5 to 15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made.

The phases of the waterfall model are shown in Figure 2-1. As you can see, the figure resembles a waterfall, in which the results of each phase flow down to the next. However, a mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made during maintenance also require backing up to earlier phases. Taken together, these phases are also called the software development life cycle.

Although the diagram depicts distinct phases, this does not mean that developers must analyze and design a complete system before coding it. Modern software development is usually

incremental and iterative. This means that analysis and design may produce a rough draft, skeletal version, or prototype of a system for coding, and then back up to earlier phases to fill in more details after some testing. For purposes of introducing this process, however, we treat these phases as distinct.

Programs rarely work as hoped the first time they are run; hence, they should be subjected to extensive and careful testing. Many people think that testing is an activity that applies only to the implementation and integration phases; however, you should scrutinize the outputs of each phase carefully. Keep in mind that mistakes found early are much less expensive to correct than those found late. Figure 2-2 illustrates some relative costs of repairing mistakes when found in different phases. These are not just financial costs but also costs in time and effort.

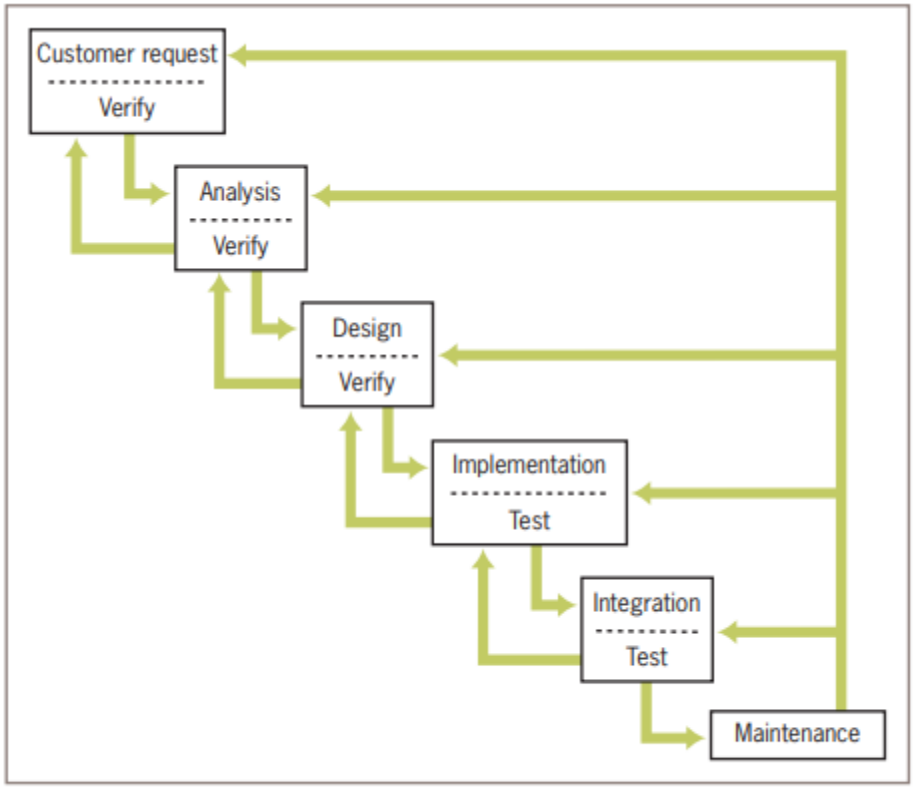

Figure 2-1 The waterfall model of the software development process

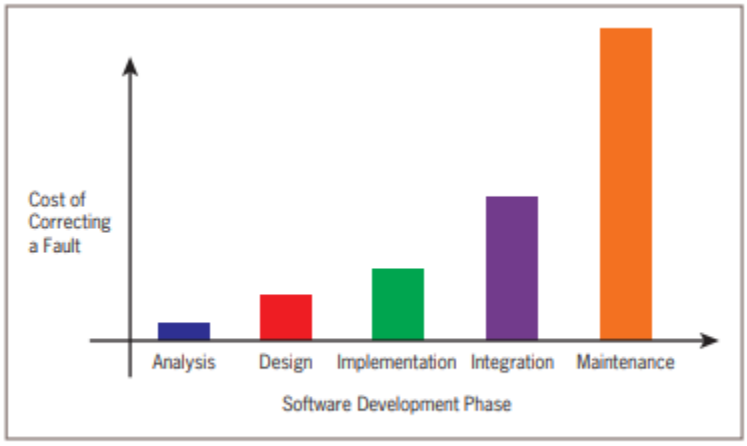

Figure 2-2 Relative costs of repairing mistakes that are found in different phases

Keep in mind that the cost of developing software is not spread equally over the phases. The percentages shown in Figure 2-3 are typical.

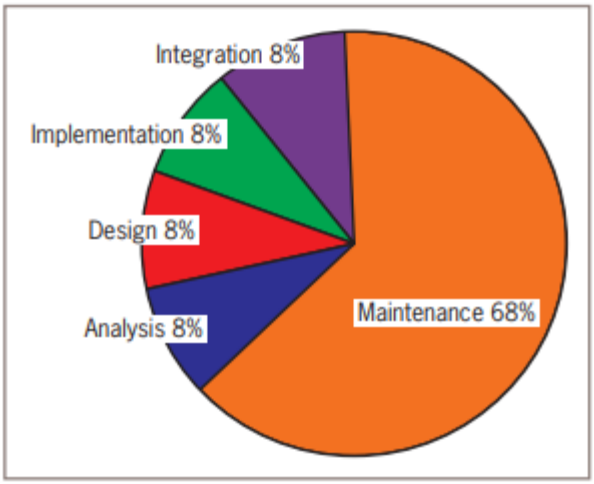

Figure 2-3 Percentage of total cost incurred in each phase of the development process

You might think that implementation takes the most time and therefore costs the most. However, as you can see in Figure 2-3, maintenance is the most expensive part of software development. The

Basic coding skills - Strings, Assignment, and Comments

Text processing is by far the most common application of computing. E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters.

Data Types

When we use data in a computer program, we do need to keep in mind the type of data we're using. We also need to keep in mind what we can do with (what operations can be performed on) particular data.

- In programming, a data type consists of a set of values and a set of operations that can be performed on those values.
- A literal is the way a value of a data type looks to a programmer.
- The programmer can use a literal in a program to mention a data value.
- When the Python interpreter evaluates a literal, the value it returns is simply that literal. Table 2-2 shows example literals of several Python data types.

Type of Data	Python Type Name	Example Literals
Integers	`int`	`-1, 0, 1, 2`
Real numbers	`float`	`-0.55, .3333, 3.14, 6.0`
Character strings	`str`	`"Hi", "", 'A', "66"`

Table 2-2 Literals for some Python data types

The first two data types listed in Table 2-2, int and float, are called numeric data types, because they represent numbers. You'll learn more about numeric data types later in this chapter. For now, we will focus on character strings—which are often referred to simply as strings.

String Literals

In Python, a string literal is a sequence of characters enclosed in single or double quotation marks. The following session with the Python shell shows some example strings:

```
>>> 'Hello there!'
'Hello there!'
>>> "Hello there!"
'Hello there!'
>>> ''
''
>>> ""
''
```

The last two string literals (" and "") represent the empty string. Although it contains no characters, the empty string is a string nonetheless. Note that the empty string is different from a string that contains a single blank space character, " ".

Double-quoted strings are handy for composing strings that contain single quotation marks or apostrophes. Here is a self-justifying example:

```
>>> "I'm using a single quote in this string!"
"I'm using a single quote in this string!"
>>> print("I'm using a single quote in this string!")
I'm using a single quote in this string!
```

Note that the print function displays the nested quotation mark but not the enclosing quotation marks. A double quotation mark can also be included in a string literal if one uses the single quotation marks to enclose the literal. When you write a string literal in Python code that will be displayed on the screen as output, you need to determine whether you want to output the string as a single line or as a multiline paragraph. If you want to output the string as a single line, you have to include the entire string literal (including its opening and closing quotation marks) in the same line of code.

Otherwise, a syntax error will occur. To output a paragraph of text that contains several lines, you could use a separate print function call for each line. However, it is more convenient to enclose the entire string literal, line breaks and all, within three consecutive quotation marks (either single or double) for printing. The next session shows how this is done:

```
>>> print("""This very long sentence extends
all the way to the next line.""")
This very long sentence extends
all the way to the next line.
```

Note that the first line in the output ends exactly where the first line ends in the code. When you evaluate a string in the Python shell without the print function, you can see the literal for the newline character, \n, embedded in the result, as follows:

```
>>> """This very long sentence extends
all the way to the next line."""
'This very long sentence extends\nall the way to the next line.'
```

Escape Sequences

The newline character \n is called an escape sequence. Escape sequences are the way Python expresses special characters, such as the tab, the newline, and the backspace (delete key), as literals. Table 2-3 lists some escape sequences in Python.

Escape Sequence	Meaning
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	The \ character
\'	Single quotation mark
\"	Double quotation mark

Table 2-3 Some escape sequences in Python

Because the backslash is used for escape sequences, it must be escaped to appear as a literal character in a string. Thus, print('\\') would display a single \ character.

String Concatenation

You can join two or more strings to form a new string using the concatenation operator +. Here is an example:

```
>>> "Hi " + "there, " + "Ken!"
'Hi there, Ken!'
```

The * operator allows you to build a string by repeating another string a given number of times. The left operand is a string, and the right operand is an integer. For example, if you want the string "Python" to be preceded by 10 spaces, it would be easier to use the * operator with 10 and one space than to enter the 10 spaces by hand. The next session shows the use of the * and + operators to achieve this result:

```
>>> " " * 10 + "Python"
'          Python'
```

Variables and the Assignment Statement

A variable associates a name with a value, making it easy to remember and use the value later in a program. You need to be mindful of a few rules when choosing names for your variables.

- For example, some names, such as if, def, and import, are reserved for other purposes and thus cannot be used for variable names.
- A variable name must begin with either a letter or an underscore (_), and can contain any number of letters, digits, or other underscores.
- Python variable names are case sensitive; thus, the variable WEIGHT is a different name from the variable weight.
- Python programmers typically use lowercase letters for variable names.
- If variable names that consist of more than one word, it's common to begin each word in the variable name (except for the first one) with an uppercase letter. This makes the variable name easier to read. For example, the name interestRate is slightly easier to read than the name interestrate.

Programmers use all uppercase letters for the names of variables that contain values that the program never changes. Such variables are known as symbolic constants. Examples of symbolic constants in the tax calculator case study are TAX_RATE and STANDARD_DEDUCTION.

Variables receive their initial values and can be reset to new values with an assignment statement. The simplest form of an assignment statement is the following:

```
<variable name> = <expression>
```

the terms enclosed in angle brackets name or describe a part of a Python code construct. Thus, the notation stands for any Python variable name, such as totalIncome or taxRate. The notation stands for any Python expression, such as " " * 10 + "Python".

The Python interpreter first evaluates the expression on the right side of the assignment symbol and then binds the variable name on the left side to this value. When this happens to the variable name for the first time, it is called defining or initializing the variable. Note that the = symbol means assignment, not equality. After you initialize a variable, subsequent uses of the variable name in expressions are known as variable references.

When the interpreter encounters a variable reference in any expression, it looks up the associated value. If a name is not yet bound to a value when it is referenced, Python signals an error. The next session shows some definitions of variables and their references:

```
>>> firstName = "Ken"
>>> secondName = "Lambert"
>>> fullName = firstName + " " + secondName
```

```
>>> fullName
'Ken Lambert'
```

The first two statements initialize the variables firstName and secondName to string values. The next statement references these variables, concatenates the values referenced by the variables to build a new string, and assigns the result to the variable fullName (“concatenate” means “glue together”). The last line of code is a simple reference to the variable fullName, which returns its value.

Variables serve two important purposes in a program. They help the programmer keep track of data that change over time. They also allow the programmer to refer to a complex piece of information with a simple name. Any time you can substitute a simple thing for a more complex one in a program, you make the program easier for programmers to understand and maintain. Such a process of simplification is called abstraction, and it is one of the fundamental ideas of computer science.

The wise programmer selects names that inform the human reader about the purpose of the data. This, in turn, makes the program easier to maintain and troubleshoot. A good program not only performs its task correctly but it also reads like an essay in which each word is carefully chosen to convey the appropriate meaning to the reader. For example, a program that creates a payment

schedule for a simple interest loan might use the variables rate, initialAmount, currentBalance, and interest.

Program Comments and Docstrings

We conclude this subsection on strings with a discussion of program comments.

- A comment is a piece of program text that the computer ignores but that provides useful documentation to programmers.
- At the very least, the author of a program can include his or her name and a brief statement about the program's purpose at the beginning of the program file.
- This type of comment, called a docstring, is a multi-line string of the form discussed earlier in this section. Here is a docstring that begins a typical program for a lab session:

```
"""
Program: circle.py
Author: Ken Lambert
Last date modified: 10/10/17

The purpose of this program is to compute the area of a
circle. The input is an integer or floating-point number
representing the radius of the circle. The output is a
floating-point number labeled as the area of the circle.
"""
```

- In addition to docstrings, end-of-line comments can document a program.
- These comments begin with the # symbol and extend to the end of a line.
- An end-of-line comment might explain the purpose of a variable or the strategy used by a piece of code, if it is not already obvious. Here is an example:

```
>>> RATE = 0.85 # Conversion rate for Canadian to US dollars
```

In a program, good documentation can be as important as executable code. Ideally, program code is self-documenting, so a human reader can instantly understand it. However, a program is often read by people who are not its authors, and even the authors might find their own code inscrutable after months of not seeing it. The trick is to avoid documenting code that has an obvious meaning, but to aid the poor reader when the code alone might not provide sufficient understanding.

With this end in mind, it's a good idea to do the following:

1. Begin a program with a statement of its purpose and other information that would help orient a programmer called on to modify the program at some future date.

2. Accompany a variable definition with a comment that explains the variable's purpose.

3. Precede major segments of code with brief comments that explain their purpose. The case study program presented earlier in this chapter does this.

4. Include comments to explain the workings of complex or tricky sections of code.

Numeric Data Types and Character Sets

The first applications of computers were created to crunch numbers. Although text and media processing have lately been of increasing importance, the use of numbers in many applications is still very important. In this section, we give a brief overview of numeric data types and their cousins, character sets.

Integers

- As you learned in mathematics, the integers include 0, the positive whole numbers, and the negative whole numbers.
- Integer literals in a Python program are written without commas, and a leading negative sign indicates a negative value.
- Although the range of integers is infinite, a real computer's memory places a limit on the magnitude of the largest positive and negative integers.
- The most common implementation of the int data type in many programming languages consists of the integers from 2 2 2,147,483,648 (2) 31 to 2,147,483,647 (2 2 1) 31 .
- However, the magnitude of a Python integer is much larger and is limited only by the memory of your computer. As an experiment, try evaluating the expression 2147483647 ** 100, which raises the largest positive int value to the 100th power. You will see a number that contains many lines of digits!

Floating-Point Numbers

- A real number in mathematics, such as the value of p (3.1416...), consists of a whole number, a decimal point, and a fractional part.
- Real numbers have infinite precision, which means that the digits in the fractional part can continue forever.
- Like the integers, real numbers also have an infinite range.
- However, because a computer's memory is not infinitely large, a computer's memory limits not only the range but also the precision that can be represented for real numbers.
- Python uses floating-point numbers to represent real numbers.
- Values of the most common implementation of Python's float type range from approximately 210308 to 10308 and have 16 digits of precision.

A floating-point number can be written using either ordinary **decimal notation** or **scientific notation.** Scientific notation is often useful for mentioning very large numbers. Table 2-4 shows some equivalent values in both notations.

Decimal Notation	Scientific Notation	Meaning
3.78	3.78e0	3.78×10^{0}
37.8	3.78e1	3.78×10^{1}
3780.0	3.78e3	3.78×10^{3}
0.378	3.78e–1	3.78×10^{-1}
0.00378	3.78e–3	3.78×10^{-3}

Table 2-4 Decimal and scientific notations for floating-point numbers

Character Sets

Some programming languages use different data types for strings and individual characters. In Python, character literals look just like string literals and are of the string type. To mark the difference in this book, we use single quotes to enclose single-character strings, and double quotes to enclose multi-character strings. Thus, we refer to 'H' as a character and "Hi!" as a string, even though they are both technically Python strings.

All data and instructions in a program are translated to binary numbers before being run on a real computer. To support this translation, the characters in a string each map to an integer value. This mapping is defined in character sets, among them the ASCII set and the Unicode set. (The term ASCII stands for American Standard Code for Information Interchange.)

- In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 127. An example of a control character is Control1D, which is the command to terminate a shell window.
- As new function keys and some international characters were added to keyboards, the ASCII set doubled in size to 256 distinct values in the mid-1980s.
- Then, when characters and symbols were added from languages other than English, the Unicode set was created to support 65,536 values in the early 1990s.
- Unicode supports more than 128,000 values at the present time.

Table 2-5 shows the mapping of character values to the first 128 ASCII codes. The digits in the left column represent the leftmost digits of an ASCII code, and the digits in the top row are the rightmost digits. Thus, the ASCII code of the character 'R' at row 8, column 2 is 82.

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	I	j	k	l	m
11	n	o	P	q	r	S	t	u	v	w
12	X	y	z	{	|	}	~	DEL		

Table 2-5 The original ASCII character set

Some might think it odd to include characters in a discussion of numeric types. However, as you can see, the ASCII character set maps to a set of integers.

- Python's **ord** and **chr** functions convert characters to their numeric ASCII codes and back again, respectively. The next session uses the following functions to explore the ASCII system:

```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(66)
'B'
```

Note that the ASCII code for 'B' is the next number in the sequence after the code for 'A'. These two functions provide a handy way to shift letters by a fixed amount. For example, if you want to shift three places to the right of the letter 'A', you can write chr(ord('A') + 3).

Expressions

As we have seen, a literal evaluates to itself, whereas a variable reference evaluates to the variable's current value. Expressions provide an easy way to perform operations on data values to produce other data values. You saw strings used in expressions earlier. When entered at the Python shell prompt, an expression's operands are evaluated, and its operator is then applied to these values to compute the value of the expression.

Arithmetic Expressions

An arithmetic expression consists of operands and operators combined in a manner that is already familiar to you from learning algebra. Table 2-6 shows several arithmetic operators and gives examples of how you might use them in Python code.

Operator	Meaning	Syntax
-	Negation	`-a`
**	Exponentiation	`a ** b`
*	Multiplication	`a * b`
/	Division	`a / b`
//	Quotient	`a // b`
%	Remainder or modulus	`a % b`
+	Addition	`a + b`
-	Subtraction	`a - b`

Table 2-6 Arithmetic operators

In algebra, you are probably used to indicating multiplication like this: ab. However, in Python, we must indicate multiplication explicitly, using the multiplication operator (*), like this: a * b. Binary operators are placed between their operands (a * b, for example), whereas unary operators are placed before their operands (–a, for example).

The precedence rules you learned in algebra apply during the evaluation of arithmetic expressions in Python:

• Exponentiation has the highest precedence and is evaluated first.

• Unary negation is evaluated next, before multiplication, division, and remainder.

• Multiplication, both types of division, and remainder are evaluated before addition and subtraction.

• Addition and subtraction are evaluated before assignment.

• With two exceptions, operations of equal precedence are left associative, so they are evaluated from left to right. Exponentiation and assignment operations are right associative, so consecutive instances of these are evaluated from right to left.

• You can use parentheses to change the order of evaluation

Table 2-7 shows some arithmetic expressions and their values.

Expression	Evaluation	Value
`5 + 3 * 2`	`5 + 6`	`11`
`(5 + 3) * 2`	`8 * 2`	`16`
`6 % 2`	`0`	`0`
`2 * 3 ** 2`	`2 * 9`	`18`
`-3 ** 2`	`-(3 ** 2)`	`-9`
`(3) ** 2`	`9`	`9`
`2 ** 3 ** 2`	`2 ** 9`	`512`
`(2 ** 3) ** 2`	`8 ** 2`	`64`
`45 / 0`	`Error: cannot divide by 0`	
`45 % 0`	`Error: cannot divide by 0`	

Table 2-7 Some arithmetic expressions and their values

The last two lines of Table 2-7 show attempts to divide by 0, which result in an error. These expressions are good illustrations of the difference between syntax and semantics. Syntax is the set of rules for constructing well-formed expressions or sentences in a language. Semantics is the set of rules that allow an agent to interpret the meaning of those expressions or sentences. A computer generates a syntax error when an expression or sentence is not well formed. A semantic error is detected when the action that an expression describes cannot be carried out, even though that expression is syntactically correct. Although the expressions 45 / 0 and 45 % 0 are syntactically correct, they are meaningless, because a computing agent cannot carry them out. Human beings can tolerate all kinds of syntax errors and semantic errors when they converse in natural languages. By contrast, computing agents can tolerate none of these errors.

With the exception of exact division, when both operands of an arithmetic expression are of the same numeric type (int or float), the resulting value is also of that type. When each operand is of a different type, the resulting value is of the more general type. Note that the float type is more general than the int type. The quotient operator // produces an integer quotient, whereas the exact division operator / always produces a float. Thus, 3 // 4 produces 0, whereas 3 / 4 produces .75.

Although spacing within an expression is not important to the Python interpreter, programmers usually insert a single space before and after each operator to make the code easier for people to read. Normally, an expression must be completed on a single line of Python code. When an expression becomes long or complex, you can move to a new line by placing a backslash character \ at the end of the current line. The next example shows this technique:

```
>>> 3 + 4 * \
2 ** 5
131
```

Make sure to insert the backslash before or after an operator. If you break lines in this manner in IDLE, the editor automatically indents the code properly.

Mixed-Mode Arithmetic and Type Conversions

You have seen how the // operator produces an integer result and the / operator produces a floating-point result with two integers. What happens when one operand is an int and the other is a float? When working with a handheld calculator, you do not give much thought to the fact that you intermix integers and floating-point numbers.

- Performing calculations involving both integers and floating-point numbers is called **mixed-mode arithmetic**.

For instance, if a circle has radius 3, you compute the area as follows:

```
>>> 3.14 * 3 ** 2
28.26
```

How does Python perform this type of calculation? In a binary operation on operands of different numeric types, the less general type (int) is temporarily and automatically converted to the more general type (float) before the operation is performed. Thus, in the example expression, the value 9 is converted to 9.0 before the multiplication. You must use a type conversion function when working with the input of numbers. A type conversion function is a function with the same name as the data type to which it converts. Because the input function returns a string as its value, you must use the function int or float to convert the string to a number before performing arithmetic, as in the following example:

```
>>> radius = input("Enter the radius: ")
Enter the radius: 3.2
>>> radius
'3.2'
>>> float(radius)
3.2
>>> float(radius) ** 2 * 3.14
32.153600000000004
```

Table 2-8 lists some common type conversion functions and their uses.

Note that the int function converts a float to an int by truncation, not by rounding to the nearest whole number. Truncation simply chops off the number's fractional part. The round function rounds a float to the nearest int as in the next example:

```
>>> int(6.75)
6
>>> round(6.75)
7
```


Conversion Function	Example Use	Value Returned
int(<a number or a string>)	int(3.77)	3
	int("33")	33
float(<a number or a string>)	float(22)	22.0
str(<any value>)	str(99)	'99'

Table 2-8 Type conversion functions

Another use of type conversion occurs in the construction of strings from numbers and other strings. For instance, assume that the variable profit refers to a floating-point number that represents an amount of money in dollars and cents. Suppose that, to build a string that represents this value for output, we need to concatenate the $ symbol to the value of profit. However, Python does not allow the use of the + operator with a string and a number:

```
>>> profit = 1000.55
>>> print('$' + profit)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

To solve this problem, we use the str function to convert the value of profit to a string and then concatenate this string to the $ symbol, as follows:

```
>>> print('$' + str(profit))
$1000.55
```

Python is a strongly typed programming language. The interpreter checks data types of all operands before operators are applied to those operands. If the type of an operand is not appropriate, the interpreter halts execution with an error message. This error checking prevents a program from attempting to do something that it cannot do.

Using Functions and Modules

Thus far in this chapter, we have examined two ways to manipulate data within expressions. We can apply an operator such as + to one or more operands to produce a new data value.

Alternatively, we can call a function such as round with one or more data values to produce a new data value. Python includes many useful functions, which are organized in libraries of code called modules.

Calling Functions: Arguments and Return Values

A function is a chunk of code that can be called by name to perform a task. Functions often require arguments, that is, specific data values, to perform their tasks. Names that refer to arguments are also known as **parameters.** When a function completes its task (which is usually some kind of computation), the function may send a result back to the part of the program that called that function in the first place. The process of sending a result back to another part of a program is known as **returning a value.**

- For example, the argument in the function call round(6.5) is the value 6.5, and the value returned is 7.
- When an argument is an expression, it is first evaluated, and then its value is passed to the function for further processing. For instance, the function call abs(4 – 5) first evaluates the expression 4 – 5 and then passes the result, –1, to abs. Finally, abs returns 1.
- The values returned by function calls can be used in expressions and statements. For example, the function call print(abs(4 – 5) + 3) prints the value 4.

Some functions have only optional arguments, some have required arguments, and some have both **required and optional arguments**. For example, the round function has one required argument, the number to be rounded. When called with just one argument, the round function exhibits its default behavior, which is to return the nearest whole number with a fractional part of 0. However, when a second, optional argument is supplied, this argument, a number, indicates the number of places of precision to which the first argument should be rounded. For example, round(7.563, 2) returns 7.56

To learn how to use a function's arguments, consult the documentation on functions in the shell. For example, Python's help function displays information about round, as follows:

```
>>> help(round)
Help on built-in function round in module builtin:

round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the same type as
    number, ndigits may be negative.
```

Each argument passed to a function has a specific data type. When writing code that involves functions and their arguments, you need to keep these data types in mind. A program that

attempts to pass an argument of the wrong data type to a function will usually generate an error. For example, one cannot take the square root of a string, but only of a number. Likewise, if a function call is placed in an expression that expects a different type of operand than that returned by the function, an error will be raised. If you're not sure of the data type associated with a particular function's arguments, read the documentation.

The math Module

Functions and other resources are coded in components called **modules.** Functions like abs and round from the __builtin__ module are always available for use, whereas the programmer must explicitly import other functions from the modules where they are defined. The math module includes several functions that perform basic mathematical operations. The next code session imports the math module and lists a directory of its resources:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

This list of function names includes some familiar trigonometric functions as well as Python's most exact estimates of the constants p and e.

- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource.
- For example, to use the value of pi from the math module, you would write the following code: math.pi. The next session uses this technique to display the value of p and the square root of 2:

```
>>> math.pi
3.1415926535897931
>>> math.sqrt(2)
1.4142135623730951
```

Once again, help is available if needed:

```
>>> help(math.cos)
Help on built-in function cos in module math:
cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

Alternatively, you can browse through the documentation for the entire module by entering help(math). The function help uses a module's own docstring and the docstrings of all its functions to print the documentation.

- If you are going to use only a couple of a module's resources frequently, you can avoid the use of the qualifier with each reference by importing the individual resources, as follows:

```
>>> from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
```

- Programmers occasionally import all of a module's resources to use without the qualifier. For example, the statement from math import * would import all of the math module's resources.

Generally, the first technique of importing resources (that is, importing just the module's name) is preferred. The use of a module qualifier not only reminds the reader of a function's purpose but also helps the computer to discriminate between different functions that have the same name.

The Main Module

In the case study, earlier in this chapter, we showed how to write documentation for a Python script. To differentiate this script from the other modules in a program (and there could be many), we call it the main module. Like any module, the main module can also be imported. Instead of launching the script from a terminal prompt or loading it into the shell from IDLE, you can start IDLE from the terminal prompt and import the script as a module. Let's do that with the taxform.py script, as follows:

```
>>> import taxform
Enter the gross income: 120000
Enter the number of dependents: 2
The income tax is $20800.0
```

After importing a main module, you can view its documentation by running the **help** function:

```
>>> help(taxform)
DESCRIPTION
Program: taxform.py
Author: Ken
Compute a person's income tax.
Significant constants
   tax rate
   standard deduction
   deduction per dependent
The inputs are
   gross income
   number of dependents
Computations:
   net income = gross income - the standard deduction - a
deduction for each dependent
   income tax = is a fixed percentage of the net income
The outputs are
   the income tax
```

Definite Iteration: The for Loop

We begin our study of control statements with repetition statements, also known as loops, which repeat an action. Each repetition of the action is known as a pass or an iteration. There are two types of loops—those that repeat an action a predefined number of times (definite iteration) and those that perform the action until the program determines that it needs to stop (indefinite iteration). In this section, we examine Python's for loop, the control statement that most easily supports definite iteration.

Executing a Statement a Given Number of Times

The form of for loop is

```
for <variable> in range(<an integer expression>):
    <statement-1>
    .
    .
    <statement-n>
```

- The first line of code in a loop is sometimes called the loop header.
- For now, the only relevant information in the header is the integer expression, which denotes the number of iterations that the loop performs.
- The colon (:) ends the loop header.
- The loop body comprises the statements in the remaining lines of code, below the header.
- These statements are executed in sequence on each pass through the loop.
- Note that the statements in the loop body must be indented and aligned in the same column.

- The IDLE shell or script window will automatically indent lines under a loop header, but you may see syntax errors if this indentation is off by even one space. It is best to indent four spaces if the indentation does not automatically occur when you move to the next line of code.

Let's explore how Python's exponentiation operator might be implemented in a loop. Recall that this operator raises a number to a given power. For instance, the expression 2 ** 3 computes the value of 23, or 2 * 2 * 2. The following session uses a loop to compute an exponentiation for a nonnegative exponent. We use three variables to designate the number, the exponent, and the product. The product is initially 1. On each pass through the loop, the product is multiplied by the number and reset to the result. To allow us to trace this process, the value of the product is also printed on each pass.

```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
        product = product * number
        print(product, end = " ")
2 4 8
>>> product
8
```

As you can see, if the exponent were 0, the loop body would not execute, and the value of product would remain as 1, which is the value of any number raised to the zero power. The use of variables in the preceding example demonstrates that our exponentiation loop is an algorithm that solves a general class of problems. The user of this particular loop not only can raise 2 to the 3rd power but also can raise any number to any nonnegative power, just by substituting different values for the variables number and exponent.

Count-Controlled Loops

When Python executes the type of for loop just discussed, it counts from 0 to the value of the header's integer expression minus 1. On each pass through the loop, the header's variable is bound to the current value of this count. The next code segment demonstrates this fact:

```
>>> for count in range(4):
        print(count, end = " ")
0 1 2 3
```

Loops that count through a range of numbers are also called count-controlled loops. The value of the count on each pass is often used in computations. For example, consider the factorial of 4, which is 1 * 2 * 3 * 4 2 5 4. A code segment to compute this value starts with a product of 1 and resets this variable to the result of multiplying it and the loop's count plus 1 on each pass, as follows:

```
>>> product = 1
>>> for count in range(4):
        product = product * (count + 1)
>>> product
24
```

Note that the value of count + 1 is used on each pass, to ensure that the numbers used are 1 through 4 rather than 0 through 3. To count from an explicit lower bound, the programmer can supply a second integer expression in the loop header. When two arguments are supplied to range, the count ranges from the first argument to the second argument minus 1. The next code segment uses this variation to simplify the code in the loop body:

```
>>> product = 1
>>> for count in range(1, 5):
        product = product * count
>>> product
24
```

The only thing in this version to be careful about is the second argument of range, which should specify an integer greater by 1 than the desired upper bound of the count. Here is the form of this version of the for loop:

```
for <variable> in range(<lower bound>, <upper bound + 1>):
    <loop body>
```

Accumulating a single result value from a series of values is a common operation in computing. Here is an example of a summation, which accumulates the sum of a sequence of numbers from a lower bound through an upper bound:

```
>>> lower = int(input("Enter the lower bound: "))
Enter the lower bound: 1
>>> upper = int(input("Enter the upper bound: "))
Enter the upper bound: 10
>>> theSum = 0
>>> for number in range(lower, upper + 1):
        theSum = theSum + number
>>> theSum
55
```

Note that we use the variable theSum rather than sum to accumulate the sum of the numbers in this code. Since sum is the name of a built-in Python function, it's a good idea to avoid using such names for other purposes in our code.

Augmented Assignment

Expressions such as x = x + 1 or x = x + 2 occur so frequently in loops that Python includes abbreviated forms for them. The assignment symbol can be combined with the arithmetic and

concatenation operators to provide augmented assignment operations. Following are several examples:

```
a = 17
s = "hi"
a += 3          # Equivalent to a = a + 3
a -= 3          # Equivalent to a = a - 3
a *= 3          # Equivalent to a = a * 3
a /= 3          # Equivalent to a = a / 3
a %= 3          # Equivalent to a = a % 3
s += " there"   # Equivalent to s = s + " there"
```

All these examples have the format

```
<variable> <operator>= <expression>
```

which is equivalent to

```
<variable> = <variable> <operator> <expression>
```

Note that there is no space between <operator> and =. The augmented assignment operations and the standard assignment operations have the same precedence.

Loop Errors: Off-by-One Error

The for loop is not only easy to write but also fairly easy to write correctly. Once we get the syntax correct, we need to be concerned about only one other possible error: The loop fails to perform the expected number of iterations. Because this number is typically off by one, the error is called an off-by-one error. For the most part, off-by-one errors result when the programmer incorrectly specifies the upper bound of the loop. The programmer might intend the following loop to count from 1 through 4, but it counts from 1 through 3:

```
# Count from 1 through 4, we think
>>> for count in range(1,4):
    print(count)
1
2
3
```

Note that this is not a syntax error, but rather a logic error. Unlike syntax errors, logic errors are not detected by the Python interpreter, but only by the eyes of a programmer who carefully inspects a program's output.

Traversing the Contents of a Data Sequence

Although we have been using the for loop as a simple count-controlled loop, the loop itself visits each number in a sequence of numbers generated by the range function. The next code segment shows what these sequences look like:

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 5))
[1, 2, 3, 4]
```

In this example, the sequence of numbers generated by the function range is fed to Python's list function, which returns a special type of sequence called a list. Strings are also sequences of characters. The values contained in any sequence can be visited by running a for loop, as follows:

```
for <variable> in <sequence>:
    <do something with variable>
```

On each pass through the loop, the variable is bound to or assigned the next value in the sequence, starting with the first one and ending with the last one. The following code segment traverses or visits all the elements in two sequences and prints the values contained in them, separated by spaces:

```
>>> for number in [6, 4, 8]:
        print(number, end = " ")
6 4 8
>>> for character in "Hi there!":
        print(character, end = " ")
H i  t h e r e !
```

Specifying the Steps in the Range

The count-controlled loops we have seen thus far count through consecutive numbers in a series. However, in some programs we might want a loop to skip some numbers, perhaps visiting every other one or every third one. A variant of Python's range function expects a third argument that allows you to nicely skip some numbers. The third argument specifies a step value, or the interval between the numbers used in the range, as shown in the examples that follow:

```
>>> list(range(1, 6, 1))   # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))   # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))   # Use every third number
[1, 4]
```

Now, suppose you had to compute the sum of the even numbers between 1 and 10. Here is the code that solves this problem:

```
>>> theSum = 0
>>> for count in range(2, 11, 2):
        theSum += count
>>> theSum
30
```

Loops That Count Down

All of our loops until now have counted up from a lower bound to an upper bound. Once in a while, a problem calls for counting in the opposite direction, from the upper bound down to the lower bound. For example, when the top-10 singles tunes are released, they might be presented in order from lowest (10th) to highest (1st) rank. In the next session, a loop displays the count from 10 down to 1 to show how this would be done:

```
>>> for count in range(10, 0, -1):
        print(count, end = " ")
10 9 8 7 6 5 4 3 2 1
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

When the step argument is a negative number, the range function generates a sequence of numbers from the first argument down to the second argument plus 1. Thus, in this case, the first argument should express the upper bound, and the second argument should express the lower bound minus 1.

Formatting Text for Output

Many data-processing applications require output that has a tabular format, like that used in spreadsheets or tables of numeric data. In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified.

- A column of data is left-justified if its values are vertically aligned beginning with their leftmost characters.
- A column of data is right-justified if its values are vertically aligned beginning with their rightmost characters.
- To maintain the margins between columns of data, left-justification requires the addition of spaces to the right of the datum, whereas right-justification requires adding spaces to the left of the datum.
- A column of data is centered if there are an equal number of spaces on either side of the data within that column.

The total number of data characters and additional spaces for a given datum in a formatted string is called its **field width.**

The print function automatically begins printing an output datum in the first available column. The next example, which displays the exponents 7 through 10 and the values of 107 through 1010, shows the format of two columns produced by the print function:

```
>>> for exponent in range(7, 11):
        print(exponent, 10 ** exponent)
7 10000000
8 100000000
9 1000000000
10 10000000000
```

Note that when the exponent reaches 10, the output of the second column shifts over by a space and looks ragged. The output would look neater if the left column were left-justified and the right column were right-justified. When we format floating-point numbers for output, we often would like to specify the number of digits of precision to be displayed as well as the field width. This is especially important when displaying financial data in which exactly two digits of precision are required.

Python includes a general formatting mechanism that allows the programmer to specify field widths for different types of data. The next session shows how to right-justify and left-justify the string "four" within a field width of 6:

```
>>> "%6s" % "four"              # Right justify
'  four'
>>> "%-6s" % "four"             # Left justify
'four '
```

The first line of code right-justifies the string by padding it with two spaces to its left. The next line of code left-justifies by placing two spaces to the string's right. The simplest form of this operation is the following:

```
<format string> % <datum>
```

This version contains a format string, the format operator %, and a single data value to be formatted. The format string can contain string data and other information about the format of the datum. To format the string data value in our example, we used the notation %<field width>s in the format string. When the field width is positive, the datum is right-justified; when the field width is negative, you get left-justification. If the field width is less than or equal to the datum's print length in characters, no justification is added. The % operator works with this information to build and return a formatted string.

To format integers, you use the letter d instead of s. To format a sequence of data values, you construct a format string that includes a format code for each datum and place the data values in a tuple following the % operator. The form of the second version of this operation follows:

```
<format string> % (<datum-1>, ..., <datum-n>)
```

Armed with the format operation, our powers of 10 loop can now display the numbers in nicely aligned columns. The first column is left-justified in a field width of 3, and the second column is right-justified in a field width of 12.

```
>>> for exponent in range(7, 11):
        print("%-3d%12d" % (exponent, 10 ** exponent))
7      10000000
8     100000000
9    1000000000
10  10000000000
```

The format information for a data value of type **float** has the form

%<field width>.<precision>f

where **.<precision>** is optional. The next session shows the output of a floating-point number without, and then with, a format string:

```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
```

Here is another, minimal, example of the use of a format string, which says to use a field width of 6 and a precision of 3 to format the float value 3.14:

```
>>> "%6.3f" % 3.14
' 3.140'
```

Note that Python adds a digit of precision to the string and pads it with a space to the left to achieve the field width of 6. This width includes the place occupied by the decimal point.

Selection: if and if-else Statements

- In some cases, instead of moving straight ahead to execute the next instruction, the computer might be faced with two alternative courses of action.
- The computer must pause to examine or test a condition, which expresses a hypothesis about the state of its world at that point in time.
- If the condition is true, the computer executes the first alternative action and skips the second alternative.
- If the condition is false, the computer skips the first alternative action and executes the second alternative.
- The condition in a selection statement often takes the form of a comparison. The result of the comparison is a Boolean value **True** or **False**.

The Boolean Type, Comparisons, and Boolean Expressions

Before you can test conditions in a Python program, you need to understand the Boolean data type, which is named for the nineteenth century British mathematician George Boole. The Boolean data type consists of only two data values—**true and false.** In Python, Boolean literals can be written in several ways, but most programmers prefer to use the standard values True and False.

Simple Boolean expressions consist of the Boolean values True or False, variables bound to those values, function calls that return Boolean values, or comparisons. The condition in a selection statement often takes the form of a comparison. For example, you might compare value A to value B to see which one is greater. The result of the comparison is a Boolean value. It is either true or false that value A is greater than value B. To write expressions that make comparisons, you have to be familiar with Python's comparison operators, which are listed in Table 3-2.

Comparison Operator	Meaning
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Table 3-2 The comparison operators

The following session shows some example comparisons and their values:

```
>>> 4 == 4
True
>>> 4 != 4
False
>>> 4 < 5
True
>>> 4 >= 3
True
>>> "A" < "B"
True
```

Note that == means equals, whereas = means assignment. When evaluating expressions in Python, you need to be aware of precedence—that is, the order in which operators are applied in complex expressions. The comparison operators are applied after addition but before assignment.

if-else Statements

- The if-else statement is the most common type of selection statement.
- It is also called a **two-way selection** statement, because it directs the computer to make a choice between two alternative courses of action.
- The if-else statement is often used to check inputs for errors and to respond with error messages if necessary.
- The alternative is to go ahead and perform the computation if the inputs are valid.

For example, suppose a program inputs the area of a circle and computes and outputs its radius. Legitimate inputs for this program would be positive numbers. But, by mistake, the user could still enter a zero or a negative number. Because the program has no choice but to use this value to compute the radius, it might crash (stop running) or produce a meaningless output. The next code segment shows how to use an if-else statement to locate (trap) this error and respond to it:

```
import math
area = float(input("Enter the area: "))
if area > 0:
    radius = math's(area / math.pi)
    print("The radius is", radius)
else:
    print("Error: the area must be a positive number")
```

Here is the Python syntax for the **if-else** statement:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

The condition in the if-else statement must be a Boolean expression—that is, an expression that evaluates to either true or false. The two possible actions each consist of a sequence of statements. Note that each sequence must be indented at least one space beyond the symbols if and else. Lastly, note the use of the colon (:) following the condition and the word else. Figure 3-2 shows a flow diagram of the semantics of the if-else statement. In that diagram, the diamond containing the question mark indicates the condition.

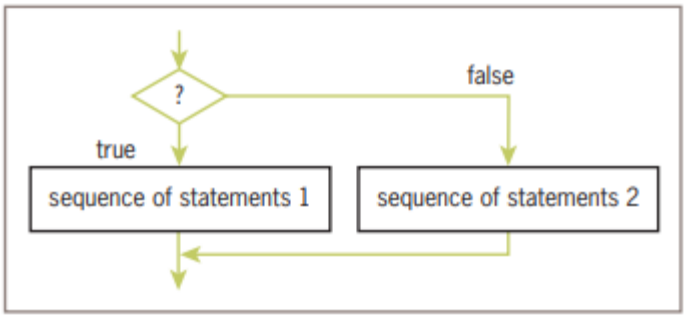

Figure 3-2 The semantics of the **if-else** statement

Our next example prints the maximum and minimum of two input numbers.

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

Python includes two functions, max and min, that make the if-else statement in this example unnecessary. In the following example, the function max returns the largest of its arguments, whereas min returns the smallest of its arguments:

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
print("Maximum:", max(first, second))
print("Minimum:", min(first, second))
```

One-Way Selection Statements

- The simplest form of selection is the if statement.
- This type of control statement is also called a **one-way selection** statement, because it consists of a condition and just a single sequence of statements.
- If the condition is True, the sequence of statements is run.
- Otherwise, control proceeds to the next statement following the entire selection statement.
- Here is the syntax for the if statement:

```
if <condition>:
    <sequence of statements>
```

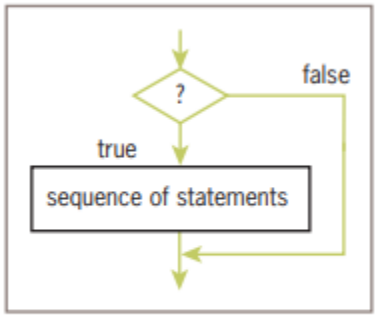

Figure 3-3 The semantics of the **if** statement

Simple if statements are often used to prevent an action from being performed if a condition is not right. For example, the absolute value of a negative number is the arithmetic negation of that

number, otherwise it is just that number. The next session uses a simple if statement to reset the value of a variable to its absolute value:

```
>>> if x < 0:
        x = -x
```

Multi-Way if Statements

- The process of testing several conditions and responding accordingly can be described in code by a multi-way selection statement.
- Here is a short Python script that uses such a statement to determine and print the letter grade corresponding to an input numeric grade:

```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

- The multi-way if statement considers each condition until one evaluates to True or they all evaluate to False.
- When a condition evaluates to True, the corresponding action is performed and control skips to the end of the entire selection statement.
- If no condition evaluates to True, then the action after the trailing else is performed. The syntax of the multi-way if statement is the following:

```
if <condition-1>:
    <sequence of statements-1>
elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

Once again, indentation helps the human reader and the Python interpreter to see the logical structure of this control statement. Figure 3-4 shows a flow diagram of the semantics of a multi-way if statement with two conditions and a trailing else clause.

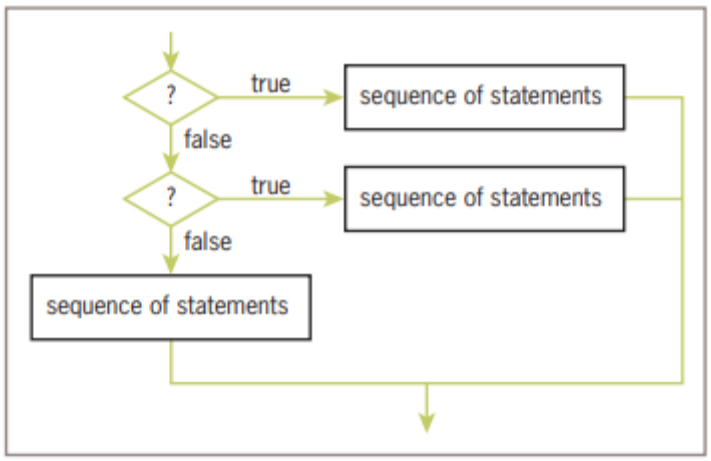

Figure 3-4 The semantics of the multi-way `if` statement

Conditional Iteration: The while Loop

Also known as loops, which repeat an action.

- Each repetition of the action is known as a **pass** or an **iteration**.
- There are two types of loops
- Those that repeat an action a predefined number of times (**definite iteration**)
- Those that perform the action until the program determines that it needs to stop (**indefinite iteration**).

The Structure and Behavior of a while Loop

- Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Such a condition is called the **loop's continuation condition.**
- If the continuation condition is false, the loop ends.
- If the continuation condition is true, the statements within the loop are executed again. The while loop is tailor-made for this type of control logic. Here is its syntax:

```
while <condition>:
    <sequence of statements>
```

The form of this statement is almost identical to that of the one-way selection statement. However, the use of the reserved word while instead of if indicates that the sequence of statements might be executed many times, as long as the condition remains true.

- Clearly, something eventually has to happen within the body of the loop to make the loop's continuation condition become false.

- Otherwise, the loop will continue forever, an error known as an **infinite loop**.
- At least one statement in the body of the loop must update a variable that affects the value of the condition. Figure 3-6 shows a flow diagram for the semantics of a while loop.

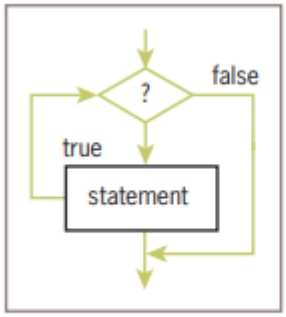

Figure 3-6 The semantics of a `while` loop

Note that there are two input statements, one just before the loop header and one at the bottom of the loop body.

- The first input statement initializes a variable to a value that the loop condition can test.
- This variable is also called the **loop control variable.**
- The second input statement obtains the other input values, including one that will terminate the loop.
- Note also that the input must be received as a string, not a number, so the program can test for an empty string. If the string is not empty, we assume that it represents a number, and we convert it to a float. Here is the Python code for this script, followed by a trace of a sample run:

```
theSum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    theSum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", theSum)

Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```

On this run, there are four inputs, including the empty string. Now, suppose we run the script again, and the user enters the empty string at the first prompt. The while loop's condition is immediately false, and its body does not execute at all! The sum prints as 0.0, which is just fine.

The while loop is also called an entry-control loop, because its condition is tested at the top of the loop. This implies that the statements within the loop can execute zero or more times.

Count Control with a while Loop

You can also use a while loop for a count-controlled loop. The next two code segments show the same summations with a for loop and a while loop, respectively.

```
# Summation with a for loop
theSum = 0
for count in range(1, 100001):
    theSum += count
print(theSum)

# Summation with a while loop
theSum = 0
count = 1
while count <= 100000:
    theSum += count
    count += 1
print(theSum)
```

Although both loops produce the same result, there is a tradeoff. The second code segment is noticeably more complex. It includes a Boolean expression and two extra statements that refer to the count variable. This loop control variable must be explicitly initialized before the loop header and incremented in the loop body. The count variable must also be examined in the explicit continuation condition. This extra manual labor for the programmer is not only time-consuming but also potentially a source of new errors in loop logic.

By contrast, a for loop specifies the control information concisely in the header and automates its manipulation behind the scenes. However, we will soon see problems for which a while loop is the only solution. Therefore, you must master the logic of while loops and also be aware of the logic errors that they could produce.

Testing

- The while loop is typically a condition-controlled loop, meaning that its continuation depends on the truth or falsity of a given condition.
- Because while loops can be the most complex control statements, to ensure their correct behavior, careful design and testing are needed.
- Testing a while loop must combine elements of testing used with for loops and with selection statements.
- Errors to rule out during testing the while loop include an incorrectly initialized loop control variable, failure to update this variable correctly within the loop, and failure to test it correctly in the continuation condition.

- Moreover, if one simply forgets to update the control variable, the result is an infinite loop, which does not even qualify as an algorithm!
- To halt a loop that appears to be hung during testing, type Control+C in the terminal window or in the IDLE shell.
- Genuine condition-controlled loops can be easy to design and test. If the continuation condition is already available for examination at loop entry, check it there and provide test data that produce 0, 1, and at least 5 iterations.
- If the loop must run at least once, use a while True loop and delay the examination of the termination condition until it becomes available in the body of the loop.
- Ensure that something occurs in the loop to allow the condition to be checked and a break statement to be reached eventually.

Break, countinue and pass

Python provides **break** and **continue** statements to handle such situations and to have good control on your loop.

The *break* Statement:

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

Example

```
for letter in'Python':# First Example
if letter =='h':
break
print'Current Letter :', letter

var=10# Second Example
whilevar>0:
print'Current variable value :',var
var=var-1
ifvar==5:
break

print"Good bye!"
```

This will produce the following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
```

Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!

The *continue* Statement:

The **continue** statement in Python returns the control to the beginning of the while loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both *while* and *for* loops.

Example:

```
for letter in'Python':# First Example
if letter =='h':
continue
print'Current Letter :', letter

var=10# Second Example
whilevar>0:
var=var-1
ifvar==5:
continue
print'Current variable value :',var
print"Good bye!"
```

This will produce following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Good bye!
```

The pass Statement:

The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

Example:

```
for letter in'Python':
if letter =='h':
pass
print'This is pass block'
print'Current Letter :', letter

print"Good bye!"
```

This will produce following result:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

The preceding code does not execute any statement or code if the value of *letter* is 'h'. The *pass* statement is helpful when you have created a code block but it is no longer required.

You can then remove the statements inside the block but let the block remain with a pass statement so that it doesn't interfere with other parts of the code.

Lazy evaluation

When using any programming language, it's important to understand when expressions are evaluated. Consider the simple expression:

```
a = b = c = 5
d = a + b * c
```

In Python, once these statements are evaluated, the calculation is immediately (or strictly) carried out, setting the value of *d* to 30. In another programming paradigm, such as in a pure functional programming language like Haskell, the value of *d* might not be evaluated until it is actually used elsewhere. The idea of deferring computations in this way is commonly known as **lazy evaluation**. Python, on the other hand, is a very strict (or eager) language. Nearly all of the time, computations and expressions are evaluated immediately. Even in the above simple expression,

the result of *b* * *c* is computed as a separate step before adding it to *a*. There are Python techniques, especially using iterators and generators, which can be used to achieve laziness. When performing very expensive computations which are only necessary some of the time, this can be an important technique in data-intensive applications.