[CHAPTER] 2

SOFTWARE DEVELOPMENT, Data Types, and Expressions

After completing this chapter, you will be able to

- Describe the basic phases of software development: analysis, design, coding, and testing
- Use strings for the terminal input and output of text
- Use integers and floating-point numbers in arithmetic operations
- Construct arithmetic expressions
- Initialize and use variables with appropriate names
- Import functions from library modules
- Call functions with arguments and use returned values appropriately
- Construct a simple Python program that performs inputs, calculations, and outputs
- Use docstrings to document Python programs

This chapter begins with a discussion of the software development process, followed by a case study in which we walk through the steps of program analysis, design, coding, and testing. We also examine the basic elements from which programs are composed. These include the data types for text and numbers and the expressions that manipulate them. The chapter concludes with an introduction to the use of functions and modules in simple programs.

2.1 The Software Development Process

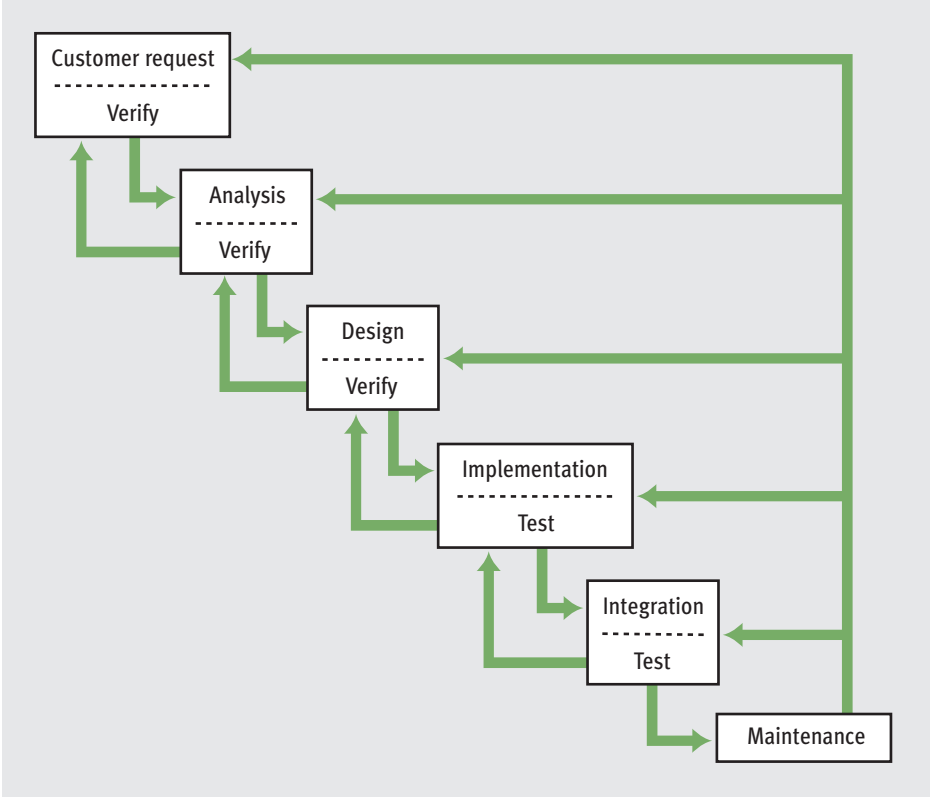
There is much more to programming than writing lines of code, just as there is more to building houses than pounding nails. The "more" consists of organization and planning, and various conventions for diagramming those plans. Computer scientists refer to the process of planning and organizing a program 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 lifespan of 5 to 15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made.

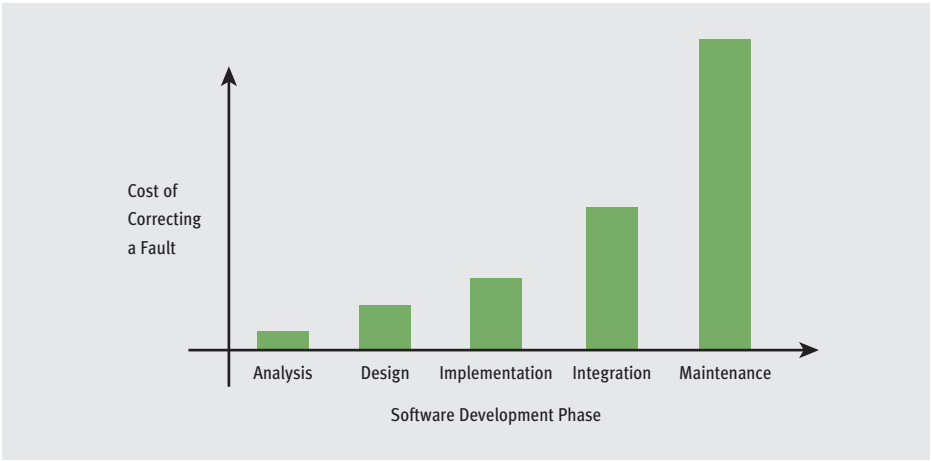
The phases of the waterfall model are shown in Figure 2.1. As you can see, the figure resembles a waterfall, in which the results of each phase flow down to the next. However, a mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made during maintenance also require backing up to earlier phases.

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.



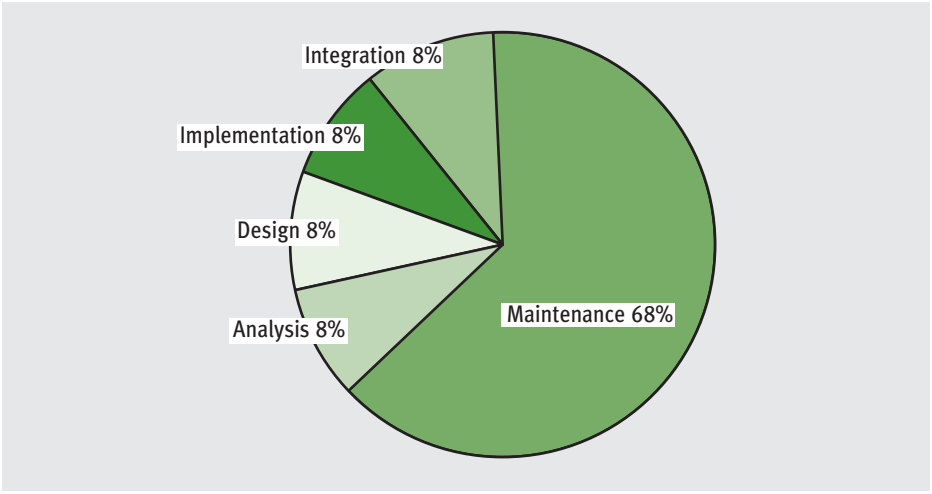
[FIGURE 2.1] The waterfall model of the software development process

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 2.2] 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 actually the most expensive part of software development. The cost of maintenance can be reduced by careful analysis, design, and implementation.

As you read this book and begin to sharpen your programming skills, you should remember two points:

1. There is more to software development than writing code.
2. If you want to reduce the overall cost of software development, write programs that are easy to maintain. This requires thorough analysis, careful design, and a good coding style. We will have more to say about coding styles throughout the book.

2.1 Exercises

1. List four phases of the software development process, and explain what they accomplish.
2. Jack says that he will not bother with analysis and design but proceed directly to coding his programs. Why is that not a good idea?

2.2 Case Study: Income Tax Calculator

Most of the chapters in this book include a case study that illustrates the software development process. This approach may seem overly elaborate for small programs, but it scales up well when programs become larger. The first case study develops a program that calculates income tax.

Each year, nearly everyone with an income faces the unpleasant task of computing his or her income tax return. If only it could be done as easily as suggested in this case study. We start with the customer request phase.

2.2.1 Request

The customer requests a program that computes a person's income tax.

2.2.2 Analysis

Analysis often requires the programmer to learn some things about the problem domain, in this case, the relevant tax law. For the sake of simplicity, let's assume the following tax laws:

- All taxpayers are charged a flat tax rate of 20%.
- All taxpayers are allowed a $10,000 standard deduction.
- For each dependent, a taxpayer is allowed an additional $3,000 deduction.
- Gross income must be entered to the nearest penny.
- The income tax is expressed as a decimal number.

Another part of analysis determines what information the user will have to provide. In this case, the user inputs are gross income and number of dependents. The program calculates the income tax based on the inputs and the tax law and then displays the income tax. Figure 2.4 shows the proposed terminal-based interface. Characters in italics indicate user inputs. The program prints the rest. The inclusion of an interface at this point is a good idea because it allows the customer and the programmer to discuss the intended program's behavior in a context understandable to both.

```
Enter the gross income: 150000.00
Enter the number of dependents: 3
The income tax is $26200.00
```

[FIGURE 2.4] The user interface for the income tax calculator

2.2.3 Design

During analysis, we specify what a program is going to do. In the next phase, design, we describe how the program is going to do it. This usually involves writing an algorithm. In Chapter 1, we showed how to write algorithms in ordinary English. In fact, algorithms are more often written in a somewhat stylized version of English called **pseudocode**. Here is the pseudocode for our income tax program:

```
Input the gross income and number of dependents
Compute the taxable income using the formula
Taxable income = gross income - 10000 - (3000 * number of dependents)
Compute the income tax using the formula
```

Tax = taxable income * 0.20
Print the tax

Although there are no precise rules governing the syntax of pseudocode, in your pseudocode you should strive to describe the essential elements of the program in a clear and concise manner. Note that this pseudocode closely resembles Python code, so the transition to the coding step should be straightforward.

2.2.4 Implementation (Coding)

Given the preceding pseudocode, an experienced programmer would now find it easy to write the corresponding Python program. For a beginner, on the other hand, writing the code can be the most difficult part of the process. Although the program that follows is simple by most standards, do not expect to understand every bit of it at first. The rest of this chapter explains the elements that make it work and much more.

```
"""
Program: taxform.py
Author: Ken Lambert

Compute a person's income tax.

1. Significant constants
       tax rate
       standard deduction
       deduction per dependent
2. The inputs are
       gross income
       number of dependents
3. Computations:
       taxable income = gross income - the standard deduction -
                    a deduction for each dependent
       income tax = is a fixed percentage of the taxable income
4. The outputs are
       the income tax
"""

# Initialize the constants
TAX_RATE = 0.20
STANDARD_DEDUCTION = 10000.0
DEPENDENT_DEDUCTION = 3000.0
```

continued

```
# Request the inputs
grossIncome = float(input("Enter the gross income: "))
numDependents = int(input("Enter the number of dependents: "))

# Compute the income tax
taxableIncome = grossIncome - STANDARD_DEDUCTION - \
                DEPENDENT_DEDUCTION * numDependents
incomeTax = taxableIncome * TAX_RATE

# Display the income tax
print("The income tax is $" + str(incomeTax))
```

2.2.5 Testing

Our income tax program can run as a script from an IDLE window. If there are no syntax errors, we will be able to enter a set of inputs and view the results. However, a single run without syntax errors and with correct outputs provides just a slight indication of a program's correctness. Only thorough testing can build confidence that a program is working correctly. Testing is a deliberate process that requires some planning and discipline on the programmer's part. It would be much easier to turn the program in after the first successful run to meet a deadline or to move on to the next assignment. But your grade, your job, or people's lives might be affected by the slipshod testing of software.

Testing can be performed easily from an IDLE window. The programmer just loads the program repeatedly into the shell and enters different sets of inputs. The real challenge is coming up with sets of inputs that can reveal an error. An error at this point, also called a **logic error** or a **design error**, is an unexpected output.

A **correct program** produces the expected output for any legitimate input. The tax calculator's analysis does not provide a specification of what inputs are legitimate, but common sense indicates that they would be numbers greater than or equal to 0. Some of these inputs will produce outputs that are less than 0, but we will assume for now that these outputs are expected. Even though the range of the input numbers on a computer is finite, testing all of the possible combinations of inputs would be impractical. The challenge is to find a smaller set of inputs, called a **test suite**, from which we can conclude that the program will likely be correct for all inputs. In the tax program, we try inputs of 0, 1, and 2 for the number of dependents. If the program works correctly with these, we can assume that it will work correctly with larger values. The test inputs for the gross income are a number equal to the standard deduction and a number twice that amount (10000 and 20000, respectively). These two values will show the cases of a minimum

expected tax (0) and expected taxes that are less than or greater than 0. The program is run with each possible combination of the two inputs. Table 2.1 shows the possible combinations of inputs and the expected outputs in the test suite.

NUMBER OF DEPENDENTS	GROSS INCOME	EXPECTED TAX
0	10000	0
1	10000	–600
2	10000	–1200
0	20000	2000
1	20000	1400
2	20000	800

[TABLE 2.1] The test suite for the tax calculator program

If there is a logic error in the code, it will almost certainly be caught using these data. Note that the negative outputs are not considered errors. We will see how to prevent such computations in the next chapter.

2.3 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. This section introduces the use of strings for the output of text and the documentation of Python programs. We begin with an introduction to data types in general.

2.3.1 Data Types

In the real world, we use data all the time without bothering to consider what kind of data we're using. For example, consider this sentence: "In 2007, Micaela paid $120,000 for her house at 24 East Maple Street." This sentence includes at least four pieces of data—a name, a date, a price, and an address—but of course you don't have to stop to think about that before you utter the sentence. You certainly don't have to stop to consider that the name consists only of text characters, the date and house price are numbers, and so on. However, 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.

2.3.2 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 multi-line 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 all the way to\nthe next line.'
>>>
```

2.3.3 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.

2.3.4 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'
>>>
```

2.3.5 Variables and the Assignment Statement

As we saw in Chapter 1, 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. In general, 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, but in the case of 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 form of an assignment statement is the following:

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

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**. The last line of code is a simple reference to the variable **fullName**, which returns its value.

Variables serve two important purposes in programs. They help the programmer keep track of data that change over the course of 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. Throughout this book, you'll learn about other abstractions used in computing, including functions, modules, and classes.

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**.

2.3.6 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 interpreter 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 purpose of the program 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: 2/10/11

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 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
```

Throughout this book, both types of documentation are colored in green.

Good documentation can be as important in a program as its 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.

2.3 Exercises

1. Let the variable **x** be **"dog"** and the variable **y** be **"cat"**. Write the values returned by the following operations:
   a. **x + y**
   b. **"the " + x + " chases the " + y**
   c. **x * 4**
2. Write a string that contains your name and address on separate lines using embedded newline characters. Then write the same string literal without the newline characters.

3 How does one include an apostrophe as a character within a string literal?

4 What happens when the **`print`** function prints a string literal with embedded newline characters?

5 Which of the following are valid variable names?

a `length`

b `_width`

c `firstBase`

d `2MoreToGo`

e `halt!`

6 List two of the purposes of program documentation.

2.4 Numeric Data Types and Character Sets

The first applications of computers were 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.

2.4.1 Integers

As you learned in mathematics, the **integers** include 0, all of the positive whole numbers, and all of 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 –2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$). However, the magnitude of a long integer can be quite large, but is still limited by the memory of your particular 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!

2.4.2 Floating-Point Numbers

A real number in mathematics, such as the value of pi (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 -10^{308} to 10^{308} 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

2.4.3 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. But they also belong to several different **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 Control+D, 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.

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)**.

2.4 Exercises

1. Which data type would most appropriately be used to represent the following data values?
   - a. The number of months in a year
   - b. The area of a circle
   - c. The current minimum wage
   - d. The approximate age of the universe (12,000,000,000 years)
   - e. Your name
2. Explain the differences between the data types **int** and **float**.
3. Write the values of the following floating-point numbers in Python's scientific notation:
   - a. 355.76
   - b. 0.007832
   - c. 4.3212
4. Consult Table 2.5 to write the ASCII values of the characters **'$'** and **'&'**.

2.5 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. 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. In this section, we examine arithmetic expressions in more detail.

2.5.1 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**, **long**, 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.

As you will see shortly, you can also break a long line of code immediately after a comma. Examples include function calls with several arguments.

2.5.2 Mixed-Mode Arithmetic and Type Conversions

When working with a handheld calculator, we do not give much thought to the fact that we 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, we compute the area as follows:

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

How do we perform a similar calculation in Python? 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.

Remember that Python has different operators for quotient and exact division. For instance,

`3 // 2 * 5.0` yields `1 * 5.0`, which yields `5.0`

whereas

`3 / 2 * 5` yields `1.5 * 5`, which yields `7.5`

In general, when you want the most precise results, you should use exact division.

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.

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

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
```

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.

2.5 Exercises

1. Let `x` = 8 and `y` = 2. Write the values of the following expressions:
   a. `x + y * 3`
   b. `(x + y) * 3`
   c. `x ** y`
   d. `x % y`
   e. `x / 12.0`
   f. `x // 6`
2. Let `x` = 4.66. Write the values of the following expressions:
   a. `round(x)`
   b. `int(x)`
3. How does a Python programmer round a `float` value to the nearest `int` value?
4. How does a Python programmer concatenate a numeric value to a string value?
5. Assume that the variable `x` has the value 55. Use an assignment statement to increment the value of `x` by 1.

2.6 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**. In this section, we examine the use of functions and modules.

2.6.1 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. 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 `float` 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.

2.6.2 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__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atanh', 'ceil','copysign', 'cos', 'cosh', 'degrees', 'e',
'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'hypot',
'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

This list of function names includes some familiar trigonometric functions as well as Python's most exact estimates of the constants π 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 π 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 interpreter to discriminate between different functions that have the same name.

2.6.3 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 Python 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.

    1. Significant constants
           tax rate
           standard deduction
           deduction per dependent
    2. The inputs are
           gross income
           number of dependents
    3. Computations:
           net income = gross income - the standard deduction -
                        a deduction for each dependent
           income tax = is a fixed percentage of the net income
    4. The outputs are
           the income tax
```

2.6.4 Program Format and Structure

This is a good time to step back and get a sense of the overall format and structure of simple Python programs. It's a good idea to structure your programs as follows:

- Start with an introductory comment stating the author's name, the purpose of the program, and other relevant information. This information should be in the form of a docstring.

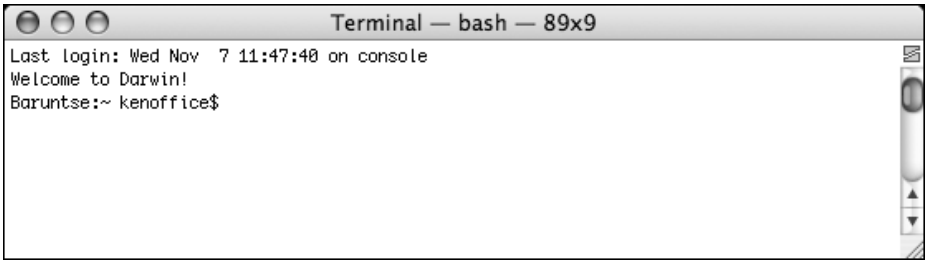
- Then, include statements that do the following:
  - Import any modules needed by the program.
  - Initialize important variables, suitably commented.
  - Prompt the user for input data and save the input data in variables.
  - Process the inputs to produce the results.
  - Display the results.

Take a moment to review the income tax program presented in the case study at the beginning of this chapter. Notice how the program conforms to this basic organization. Also, notice that the various sections of the program are separated by whitespace (blank lines). Remember, programs should be easy for other programmers to read and understand. They should read like essays!

2.6.5 Running a Script from a Terminal Command Prompt

Thus far in this book, we have been developing and running Python programs experimentally in IDLE. When a program's development and testing are finished, the program can be released to others to run on their computers. Python must be installed on a user's computer, but the user need not run IDLE to run a Python script.

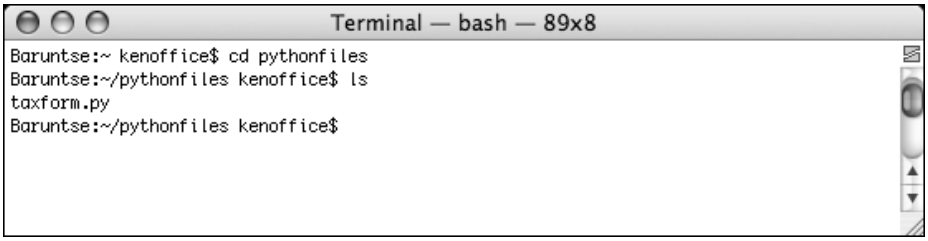
One way to run a Python script is to open a terminal command prompt window. On a computer running Windows, this is the DOS command prompt window; to open it, select the **Start** button, select **All Programs**, select **Accessories**, and then select **Command Prompt**. On a Macintosh or UNIX-based system, this is a terminal window. A terminal window on a Macintosh is shown in Figure 2.5.



[FIGURE 2.5] A terminal window on a Macintosh

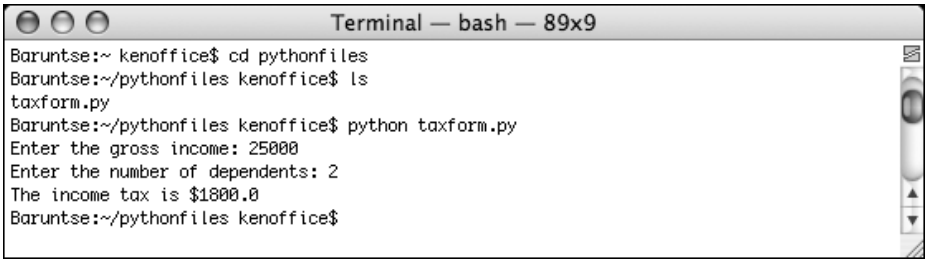
After the user has opened a terminal window, she must navigate or change directories until the prompt shows that she is attached to the directory that contains the Python script. For example, if we assume that the script named **taxform.py** is in

the **pythonfiles** directory under the terminal's current directory, Figure 2.6 shows the commands to change to this directory and list its contents.



[FIGURE 2.6] Changing to another directory and listing its contents

When the user is attached to the appropriate directory, she can run the script by entering the command **`python scriptname.py`** at the command prompt. Figure 2.7 shows this step and a run of the **`taxform`** script.



[FIGURE 2.7] Running a Python script in a terminal window

All Python installations also provide the capability of launching Python scripts by double-clicking the files from the operating system's file browser. On Windows systems, this feature is automatic, whereas on Macintosh and UNIX-based systems, the **`.py`** file type must be set to launch with the Python launcher application. When you launch a script in this manner, however, the command prompt window opens, shows the output of the script, and closes. To prevent this fly-by-window problem, you can add an input statement at the end of the script that pauses until the user presses the enter or return key, as follows:

```
input("Please press enter or return to quit the program. ")
```

2.6 Exercises

1. Explain the relationship between a function and its arguments.

2. The **`math`** module includes a **`pow`** function that raises a number to a given power. The first argument is the number, and the second argument is the exponent. Write a code segment that imports this function and calls it to print the values 8^2 and 5^4.

3. Explain how to display a directory of all of the functions in a given module.

4. Explain how to display help information on a particular function in a given module.

Summary

- The waterfall model describes the software development process in terms of several phases. Analysis determines what the software will do. Design determines how the software will accomplish its purposes. Implementation involves coding the software in a particular programming language. Testing and integration demonstrate that the software does what it is intended to do as it is put together for release. Maintenance locates and fixes errors after release and adds new features to the software.
- Literals are data values that can appear in a program. They evaluate to themselves.
- The string data type is used to represent text for input and output. Strings are sequences of characters. String literals are enclosed in pairs of single or double quotation marks. Two strings can be combined by concatenation to form a new string.
- Escape characters begin with a backslash and represent special characters such as the delete key and the newline.
- A docstring is a string enclosed by triple quotation marks and provides program documentation.
- Comments are pieces of code that are not evaluated by the interpreter but can be read by programmers to obtain information about a program.

- Variables are names that refer to values. The value of a variable is initialized and can be reset by an assignment statement. In Python, any variable can name any value.
- The **`int`** data type represents integers. The **`float`** data type represents floating-point numbers. The magnitude of an integer or a floating-point number is limited by the memory of the computer, as is the number's precision in the case of floating-point numbers.
- Arithmetic operators are used to form arithmetic expressions. Operands can be numeric literals, variables, function calls, or other expressions.
- The operators are ranked in precedence. In descending order, they are exponentiation, negation, multiplication (*, /, and % are the same), addition (+ and – are the same), and assignment. Operators with a higher precedence are evaluated before those with a lower precedence. Normal precedence can be overridden by parentheses.
- Mixed-mode operations involve operands of different numeric data types. They result in a value of the more inclusive data type.
- The type conversion functions can be used to convert a value of one type to a value of another type after input.
- A function call consists of a function's name and its arguments or parameters. When it is called, the function's arguments are evaluated, and these values are passed to the function's code for processing. When the function completes its work, it may return a result value to the caller.
- Python is a strongly typed language. The interpreter checks the types of all operands within expressions and halts execution with an error if they are not as expected for the given operators.
- A module is a set of resources, such as function definitions. Programmers access these resources by importing them from their modules.
- A semantic error occurs when the computer cannot perform the requested operation, such as an attempt to divide by 0. Python programs with semantic errors halt with an error message.
- A logic error occurs when a program runs to a normal termination but produces incorrect results.

REVIEW QUESTIONS

1 What does a programmer do during the analysis phase of software development?

a Codes the program in a particular programming language
b Writes the algorithms for solving a problem
c Decides what the program will do and determines its user interface
d Tests the program to verify its correctness

2 What must a programmer use to test a program?

a All possible sets of legitimate inputs
b All possible sets of inputs
c A single set of legitimate inputs
d A reasonable set of legitimate inputs

3 What must you use to create a multi-line string?

a A single pair of double quotation marks
b A single pair of single quotation marks
c A single pair of three consecutive double quotation marks
d Embedded newline characters

4 What is used to begin an end-of-line comment?

a `/` symbol
b `#` symbol
c `%` symbol

5 Which of the following lists of operators is ordered by decreasing precedence?

a `+, *, **`
b `*, /, %`
c `**, *, +`

6 The expression `2 ** 3 ** 2` evaluates to which of the following values?

a 64
b 512
c 8

7 The expression **round(23.67)** evaluates to which of the following values?

a 23
b 23.7
c 24.0

8 Assume that the variable **name** has the value 33. What is the value of **name** after the assignment statement **name = name * 2** executes?

a 35
b 33
c 66

9 Write an import statement that imports just the functions **sqrt** and **log** from the **math** module.

10 What is the purpose of the **dir** function and the **help** function?

PROJECTS

In each of the projects that follow, you should write a program that contains an introductory docstring. This documentation should describe what the program will do (analysis) and how it will do it (design the program in the form of a pseudocode algorithm). Include suitable prompts for all inputs, and label all outputs appropriately. After you have coded a program, be sure to test it with a reasonable set of legitimate inputs.

1 The tax calculator program of the case study outputs a floating-point number that might show more than two digits of precision. Use the **round** function to modify the program to display at most two digits of precision in the output number.

2 You can calculate the surface area of a cube if you know the length of an edge. Write a program that takes the length of an edge (an integer) as input and prints the cube's surface area as output.

3 Five Star Video rents new videos for $3.00 a night, and oldies for $2.00 a night. Write a program that the clerks at Five Star Video can use to calculate the total charge for a customer's video rentals. The program should prompt the user for the number of each type of video and output the total cost.

4 Write a program that takes the radius of a sphere (a floating-point number) as input and outputs the sphere's diameter, circumference, surface area, and volume.

5 An object's momentum is its mass multiplied by its velocity. Write a program that accepts an object's mass (in kilograms) and velocity (in meters per second) as inputs and then outputs its momentum.

6 The kinetic energy of a moving object is given by the formula $KE=(1/2)mv^2$, where m is the object's mass and v is its velocity. Modify the program you created in Project 5 so that it prints the object's kinetic energy as well as its momentum.

7 Write a program that calculates and prints the number of minutes in a year.

8 Light travels at $3 * 10^8$ meters per second. A light-year is the distance a light beam travels in one year. Write a program that calculates and displays the value of a light-year.

9 Write a program that takes as input a number of kilometers and prints the corresponding number of nautical miles. Use the following approximations:

- A kilometer represents 1/10,000 of the distance between the North Pole and the equator.
- There are 90 degrees, containing 60 minutes of arc each, between the North Pole and the equator.
- A nautical mile is 1 minute of an arc.

10 An employee's total weekly pay equals the hourly wage multiplied by the total number of regular hours plus any overtime pay. Overtime pay equals the total overtime hours multiplied by 1.5 times the hourly wage. Write a program that takes as inputs the hourly wage, total regular hours, and total overtime hours and displays an employee's total weekly pay.